High-Performance Architectures for Accelerating Sparse LU Computation

A Thesis

Submitted to the Faculty

of

Drexel University

by

Kevin Cunningham

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Engineering

June 2011

Acknowledgements

I would like to thank Dr. Prawat Nagvajara for introducing me to research and giving me the opportunity to work on multiple research projects. I appreciate the guidance and support that he has given me over the past several years. I would also like to thank Dr. Jeremy Johnson for providing valuable input and advice, especially as I worked to finish this thesis. Thank you to Dr. Mark Hempstead for passing on valuable knowledge about computer architectures and for serving on my thesis committee.

Thanks to Gavin Harrison for his help with the DPA Simulator, and to Doug Jones for creating the simulator. Thanks to Tim Chagnon for his previous work on sparse LU and for providing me with many valuable resources.

Thank you to all of my friends and family for being supportive and understanding throughout my college career. Lastly, thank you to my girlfriend, Josa, for providing advice and support throughout the thesis process.

Contents

List of Figures

List of Tables

Abstract

High-Performance Architectures for Accelerating Sparse LU Computation

Kevin Cunningham

Advisors: Prawat Nagvajara, PhD

Jeremy Johnson, PhD

Sparse Lower-Upper (LU) Triangular Decomposition is important to many different applications, including power system analysis. High-performance sparse linear algebra software packages, executing on general-purpose processors, experience lower performance when processing power system matrices. This observation motivated previous work on the design of custom hardware, implemented, in FPGA, to improve performance of sparse LU. While improved performance was obtained, significant effort was required to design and implement the hardware. This thesis investigates the combination of general purpose architectures and a hardware accelerator, for a crucial component of sparse LU, to achieve similar performance results without the design overhead. One architecture, combining a general-purpose processor with a hardware accelerator, achieves a 1.29X speedup over software for a 26K-Bus power system. The second architecture, a modification of the Data Pump Architecture, provides a 2.27X speedup over software on the 26K-bus power system. These results show that speedup for sparse LU is possible, without designing a complete custom hardware solution, using a small hardware accelerator, provided a tightly coupled architecture is available to feed data to the accelerator.

1. Introduction

1.1 Overview

General-purpose processors, Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs) have performance, power and cost tradeoffs that must be considered when selecting the appropriate platform for an application. When seeking optimal performance a combination of these platforms may be appropriate. It is a challenge to efficiently combine these different devices so that applications can take advantage of their computing strengths in a tightly-coupled, low-overhead manner. The movement of data and interaction between devices is an extremely important factor that affects the performance of these combined architectures.

In current systems, GPUs and FPGAs are typically separate, external devices from the processor. These devices operate independently and have their own memory. Depending on the external device, communication between the device and processor can occur over different types of connections, such as USB, Ethernet, HyperTransport, or PCI-Express. Figure 1.1 is an example of a CPU and FPGA connected to each other with some form of communication. While this setup can work well for applications that transfer large blocks of data and communicate infrequently between the devices, more close-coupled applications suffer from the overhead and latency of communication. The link between the devices can become a bottleneck in the system leading to poor performance.

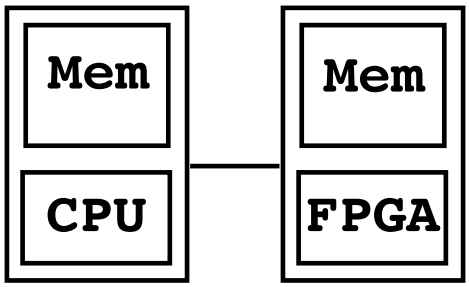

Figure 1.1: Common CPU-FPGA Architecture

Sparse Lower-Upper (LU) Triangular decomposition, particularly in power system analysis, performs poorly on general-purpose processors [2]. To address this issue, a custom sparse LU hardware solver was implemented on a FPGA, in 2005 [12]. With the hardware solver, consisting of a custom computation unit, cache unit, and interconnection to external memory, the host CPU downloaded the entire matrix data to the memory and computation unit, and afterward uploaded the resulting L and U matrices, successfully using the paradigm shown in Figure 1.1. The LU hardware solver on a Xilinx V4 LX200 133 MHz, was shown to provide a 3X performance gain over the LU software package UMFPACK 5.2.0 running on a Pentium 4 (3.2 GHz) [12]. However, with recent advances in processor technology and larger caches, the FPGA requires a faster clock frequency to obtain a similar performance gain. The LU hardware solver [12] would need to run at 500MHz to match LU software performance on the Core i7 at 3.2GHz [2].

While a custom hardware solver can improve sparse LU performance, the required design time and complexity of such a design makes it difficult to implement and maintain. In addition, the low flexibility of a custom design, compared to software, limits the ability to utilize new algorithms or methods, scale to handle larger input sizes, and port to new platforms.

A paradigm where software and hardware accelerators work together seems to be a much more attractive option. This method reduces the design time and complexity of custom hardware, because it is not necessary to design an entire hardware system. In addition, the benefits of high-performance software can still be utilized by extracting low performance areas and using an accelerator in its place. This paradigm allows for more flexibility and scalability, while being able to take advantage of improving general-purpose processor technologies.

This work investigates the performance of sparse LU decomposition on three different architectures. The first architecture combines a general-purpose processor and hardware accelerator using the paradigm, shown in Figure 1.1. This method uses a buffering strategy to efficiently utilize the high-performance transfer bus connecting the processor and FPGA. The results show the drawbacks and inefficiencies in using this method for sparse LU. The

second architecture combines a general-purpose processor with an FPGA accelerator over a low-latency local bus. The processor and FPGA share the same memory system and communicate data through direct memory access (DMA) transfers. The third architecture, the Data Pump Architecture (DPA) [17], uses processors to control the movement of data through the system and efficiently provide data to a hardware computation unit. The performance analysis shows the benefit of using the second and third architectures, over the first existing architecture, to perform the sparse LU decomposition in power flow calculations. The key to good performance is the acceleration of costly indexing operations combined with an efficient means to feed data to the accelerator.

Chapter 1 provides background on sparse LU decomposition, power flow calculation, and reconfigurable hardware. Chapter 2 describes the existing and proposed architectures combining general-purpose processors with reconfigurable hardware. Chapter 3 describes the merge hardware accelerator, which is used to increase LU performance. Chapters 5, 6, and 7 present the performance results of sparse LU on the three different architectures, and Chapter 8 compares the performance of the architectures.

1.2 Sparse LU Decomposition

The Lower-Upper (LU) decomposition factorizes a matrix, A, into a lower triangular matrix, L, and an upper triangular matrix, U, as shown in (1.2).

$$Ax = LUx = b$$

Using forward and backward substitution, the solution of a system of equations can be found from the decomposed matrix. Once the matrix is decomposed, it can be used to solve for different b vectors without factorizing again. This is useful for power systems, where the matrix A remains the same for multiple iterations of b vectors.

The matrix, A, is called sparse when it has a large number of zero elements. Sparse LU denotes LU factorization when the matrix A is sparse.

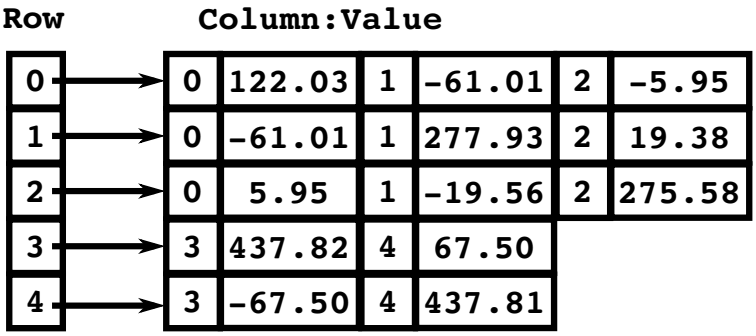

Figure 1.2: Row Compressed Format

1.2.1 Sparse Compressed Formats

When computing with sparse matrices, the matrices are often stored in a compressed format that only stores the nonzero values of the matrix. Using a compressed format reduces the memory needed to store a matrix and speeds up computation by ignoring the zero values of the matrix. One common sparse format, known as the triplet format stores the row number, column number, and value for each element in the matrix. Additional storage can be saved using compressed row or column format. Compressed row format stores each row as a list of nonzero values and column numbers. Similarly, compressed column format stores each column as a list of row numbers and values.

Row v of a sparse matrix with N columns is represented as $v = \{v_i, i = 1..NNZ(v)\}$, $v_i = (v_i.col, v_i.val)$, $v_i.col \in 1..N$, $v_i.col < v_{i+1}.col$; where $NNZ(v)$ denotes the Number of Non-Zero elements and $v_i.val$ is the value at the column number $v_i.col$. This row-compressed representation enumerates only the non-zero elements of a sparse row-vector; moreover, v is a sorted variable-length array.

A sparse matrix, in row-compressed format, is represented by an array of pointers to variable length arrays, as shown in Figure 1.2. The array of pointers is of length N, the number of rows in the matrix, and is indexed by the row number. Each pointer in the array points to an array holding the column number and value of each element in that row. A separate array of length N holds the length of each row.

One of the difficulties with sparse matrices stored in a compressed format is that the

element column number does not correspond to the inherent array index as in dense matrix storage formats. This means that the column number must be fetched from memory for operations that need to compare column numbers between rows. Fetching and comparing column numbers can slow down the performance of general-purpose processors, due to cache misses and data-dependent branching [5]. Because the branch outcome depends on the data being processed and not a predictable pattern, branch predictors do not provide any consistent benefit. Mispredicted branches can cause stalls or pipeline flushes in general-purpose processors, potentially leading to poor performance.

1.2.2 Gaussian LU Decomposition

There are multiple methods for performing the LU decomposition on a matrix. The software package UMFPACK [8], which is regarded as one of the best performing software packages for the sparse LU decomposition, uses a multi-frontal method for sparse LU. The multi-frontal method uses an elimination tree to analyze the non-zero structure of the matrix and separate the matrix into independent, frontal matrices that can be processed in parallel. The frontal matrices are dense and can take advantage of the high-performance dense software routines, such as BLAS. For some sparse matrices, the size of the frontal matrices can be small, limiting the peak performance of the BLAS routines. In addition, the overhead of analyzing the matrix structure, allocating additional memory for the frontal matrices, and combining the parallely computed parts can reduce the performance on some systems [2].

While UMFPACK performs well on many sparse matrices, previous work has discovered that a simple Gaussian elimination LU implementation outperforms UMFPACK on power system matrices [2]. Figure 1.3 shows the performance speedup, on an Intel Core i7 965 at 3.2GHz, of the Gaussian method over UMFPACK for power systems. For the smaller power systems, the additional overhead of the multi-frontal method reduces the performance of UMFPACK relative to the Gaussian method. As the power system size grows, the average size of the frontal matrices increases, allowing for better performance for UMFPACK [2].

Algorithm 1, Gaussian LU, uses both a row compressed sparse format and column

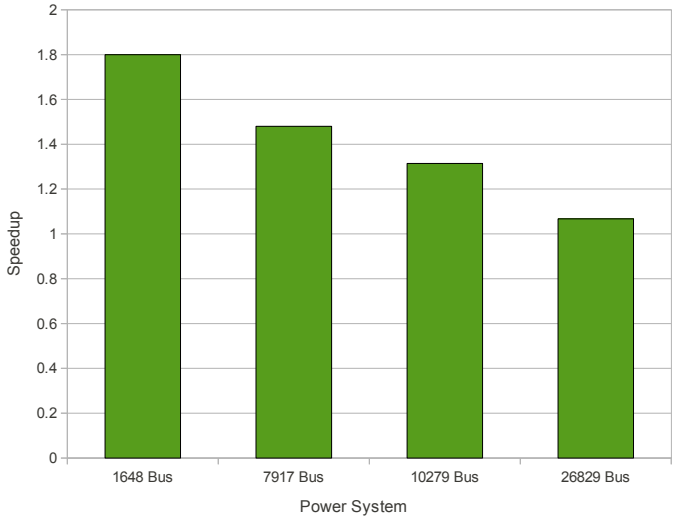

Figure 1.3: Speedup of Gaussian LU over UMFPACK on Power Systems [2]

compressed sparse format (column map) to perform Gaussian LU on a sparse matrix. The row compressed format stores the matrix values, while the column map only holds the row numbers in each column. For a matrix with N columns, the column map is an array of N variable-length arrays containing the row numbers of the non-zero elements in each column. The array of row numbers need not be sorted. The information on the row positions of the non-zero elements in a column allows fast access to the rows with non-zero elements below the diagonal position (i, i). These rows, called the submatrix rows, are accessed during the i^{th} iteration of the LU algorithm. The pivot row is selected from the submatrix by finding the maximum leading element. The pivot row is output as the i^{th} row of the U matrix.

Next, each submatrix row undergoes row reduction. It is transformed by adding into it the scaled pivot row to eliminate the leading non-zero element. With the row compressed data, the row reduction is a merge of two sorted variable-length arrays into a single sorted variable-length array. The following "Merge" algorithm describes a reduction of row j with a pivot row into a single output row.

Figure 1.4 shows an example of two sparse rows, u and v, being added into one combined row. The elements in each of the rows are ordered by column number in the output row, and elements present in the same column of both rows are added together.

Algorithm 1 Gaussian LU

```
for i = 1 → N do
  pivot_search()
  update_U()
  for j = 1 → NUM_SUBMATRIX_ROWS do
    merge(pivot_row, j)
    update_L()
    update_colmap()
  end for
end for
```

Algorithm 2 Merge

```
for p ∈ pivot_row and s ∈ row_j do
  if p.col == s.col then
    out.col ← p.col
    out.val ← p.val + s.val
    p++
    s++
  else
    if p.col < s.col then
      out.col ← p.col
      out.val ← p.val
      p++
    else
      out.col ← s.col
      out.val ← s.val
      s++
    end if
  end if
end for
```

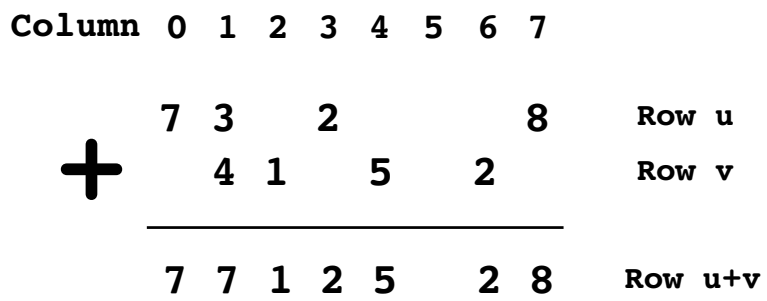

Figure 1.4: Sparse Row Addition Example

After the merge process is complete, the pivot scale factors are output to the L matrix and the column map is updated with the row numbers corresponding to the additional non-zero elements from the merge process.

The LU decomposition is complete after all elements under the main diagonal are eliminated.

1.2.3 Merge Accelerator Motivation

For power systems, merging involves mainly data movement where only 2-3% of matrix elements during LU calculation need to be added [2]. However, the comparison between column numbers leads to data-dependent branching, which does not perform efficiently on general-purpose processors. Because the outcome of the branch depends on a value in memory and not a consistent pattern, processors are unable to efficiently use branch prediction during the merge. The processor can stall or flush the pipeline, in the case of a misprediction, leading to reduced performance in many cases. In addition, the indexing overhead of accessing the column number and value of each element adds additional complexity to the merge process. The performance of the merge process on a general-purpose processor highly depends on the structure of the rows it is processing, which can vary widely [5].

The reduced performance of the merge operation on general-purpose processors indicates that a custom hardware accelerator may be able to improve the performance of the merge operation. Residing in the inner loop of the LU algorithm, the merge operation is the primary operation for sparse LU. Because of the importance of the merge operation in sparse LU, an accelerator has the potential to provide a significant increase in performance [5].

1.3 Power Systems

Electrical power is an important resource that many industries and people depend upon on a daily basis. In order to supply electrical power to the many different users, power transmission systems must transfer power from the power generation facilities to the substations that distribute power to the users. Power grid operators must analyze the power

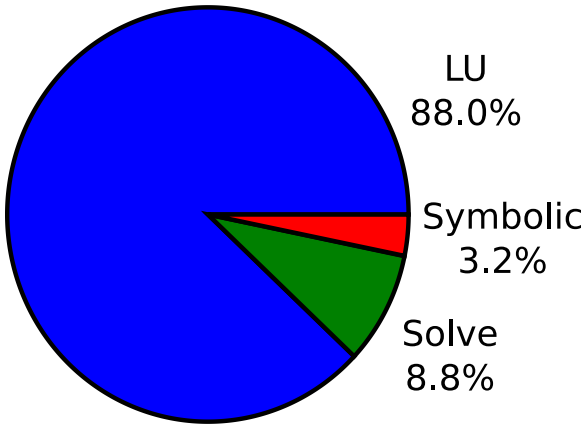

Figure 1.5: Power Flow Execution Profile [18]

grid in real time, using contingency analysis, to ensure its correct operation and protect against failures that can lead to blackouts [1].

Simulations of the power grid allow operators to estimate the current state of the system using power flow (or load flow) calculations. Using the transmission lines and connection nodes, the power flow calculation models the power system with a graph. A mathematical model is generated from the graph using Kirchoff's current laws, which can then be solved for the power flowing through each transmission line in the system. Representing the model with a sparse matrix allows for the use of high-performance sparse linear algebra computation to solve the power equations [1].

Lower-Upper (LU) triangular decomposition as part of the Newton-Raphson technique for solving power equations, is necessary for calculating power flow in contingency analysis. Based on the flow solution of each contingency, the grid operators determine whether the system is operating within its generation and transmission capability, or whether precaution actions must be taken to prevent the system from going to an unsecure state if failure occurs. Full grid analysis can require the power flow calculation to be run thousands of times for each potential failure [1].

The LU decomposition accounts for a large portion of the power flow execution time, as shown in Figure 1.5. Because of the importance of LU to the power flow calculation and the number of times that the calculation is performed, even small speedups in the LU

Table 1.1: Power System Matrix Properties

System	# Rows/Cols	NNZ	Sparsity
1648 Bus	2,982	21,196	0.238%
7917 Bus	14,508	105,522	0.050%
10278 Bus	19,285	134,621	0.036%
26829 Bus	50,092	351,200	0.014%

calculation can result in significant time savings for the full contingency analysis. This saved time allows grid operators to detect potential failures sooner and complete more analysis of the grid.

1.3.1 Power System Benchmark Data

Several benchmark power grid systems have been collected for analysis of the power flow calculation. Some of the available systems include 1,648-node 7,917-node, 10,278-node and 26,829-node power systems. The Jacobian matrices from these systems have been pre-ordered using the AMD [9] algorithm to reduce fill-in during the LU decomposition. Table 1.1 lists the properties of the four power system matrices.

The 1648-bus and 7917-bus benchmark systems are from PSS/E, Siemens Power Systems Simulator for Engineering, and the 10279-bus and 26829-bus systems are from PJM Interconnect. Figure 1.6 shows the non-zero structure of the approximate minimum-degree-ordered [9] Jacobian matrices of the systems.

1.4 Reconfigurable Hardware

Reconfigurable hardware devices, such as Field Programmable Gate Arrays (FPGA) and Complex Programmable Logic Devices (CPLD), can be configured at run-time to create a large number of possible custom hardware designs. Application Specific Integrated Circuits (ASIC) allow for higher performance and lower power consumption custom hardware designs, but are not reconfigurable. By sacrificing some performance and power efficiency, FPGAs and CPLDs are able to provide an alternative to ASICs that require less development time and cost. Reconfigurable hardware allows for a design to be easily updated to

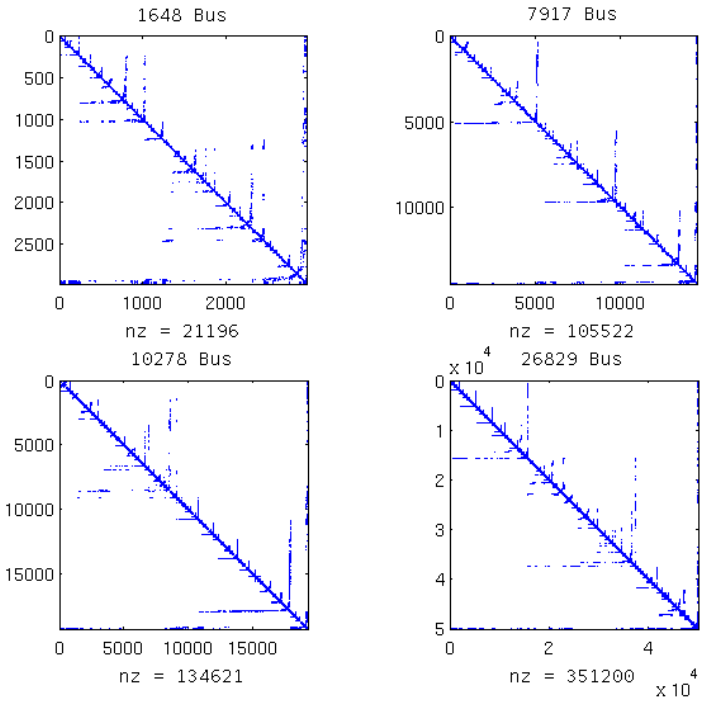

Figure 1.6: Non-Zero Patterns of AMD Pre-Ordered Power System Matrices

address errors or changing requirements, while an ASIC requires a whole new chip to be fabricated. This also benefits the development and testing process, because a design can be verified, tested, and updated much faster [21].

Another benefit of reconfigurable hardware is that a single piece of hardware can perform different functions at different times. While an ASIC design requires less area than the same design in reconfigurable hardware, the increased area of reconfigurable hardware allows for many different designs to be realized within the same area. Being able to use the same hardware for many different designs reduces production cost, because a separate piece of hardware does not have to be fabricated for each desired function.

1.4.1 CPLD

Programmable Logic Arrays (PLA) and Programmable Array Logic (PAL) are arrays of AND and OR gates that can realize boolean logic functions. The PLA has greater

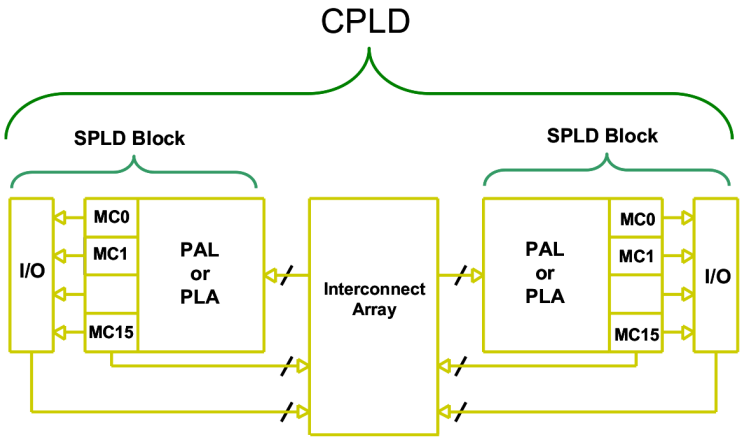

Figure 1.7: CPLD Architecture [21]

flexibility, because both the AND and OR gate arrays are configurable, while the PAL has fixed OR gates and allows configuration of the AND gates. Because the PAL has less flexibility than the PLA, it is able to achieve faster performance. Complex Programmable Logic Devices combine multiple PALs or PLAs into a single device by connecting them with an interconnect array. Figure 1.7 shows a simple CPLD architecture. The CPLD provides fast reconfigurable hardware within a small area [21].

1.4.2 FPGA

The Field Programmable Gate Array is a programmable array of logic cells with interconnections between the logic cells. A basic logic cell consists of a lookup table (LUT) and a flip-flop. Newer FPGAs have multiple LUTs and flip-flops in a single logic cell [24]. Figure 1.8 shows a simple logic cell and Figure 1.9 shows an example of the FPGA architecture, which connects the logic cells through an interconnection network [21].

FPGAs are more complex to program than CPLDs, but provide greater flexibility in the designs.

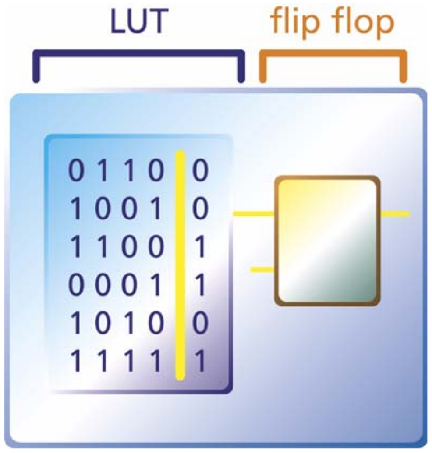

Figure 1.8: SRAM Logic Cell [21]

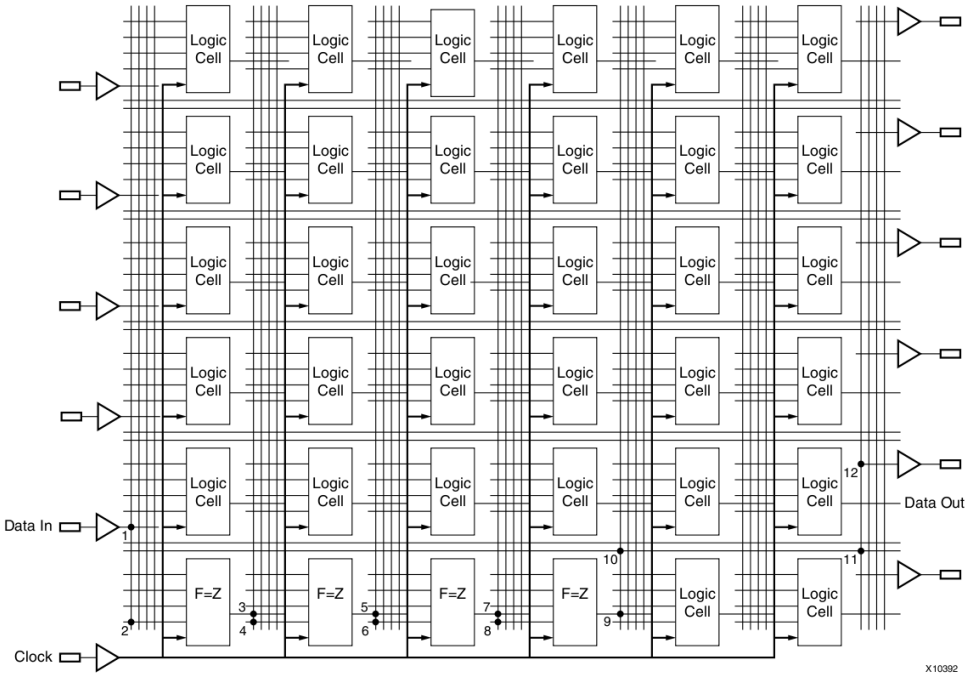

Figure 1.9: FPGA Architecture [21]

1.4.3 Reconfigurable Hardware Accelerators

Modern, general-purpose processors provide high performance and flexibility in a wide range of applications, but they do not meet the power or performance needs of some applications. In some cases, a custom design may be able to provide better performance and lower power consumption than a general-purpose processor. While a custom hardware solution can provide better performance, the development of such a design usually requires a large amount of time and effort. Therefore, it is not always a good investment to completely implement a solution with a custom hardware design. In order to save development time and effort, it can be advantageous to partition a design between general-purpose processors and custom designs allowing each part to be implemented where it will perform best.

Currently, one of the drawbacks of using reconfigurable hardware and general-purpose processors together is the lack of an architecture that effectively combines them for high bandwidth and low latency communication with low communication overhead. This communication is necessary for many data-dependent applications that require the processor and hardware to operate on the same data. One method of using an FPGA is to download an entire set of input data and then upload the output data when the computation is complete. A drawback associated with this method is that it requires implementing large, complex designs in hardware that replace the functionality of the existing processor. A second method is to have the processor send streams of data to the FPGA to process. Existing FPGAs can receive data from a processor over connections such as USB, PCI-Express, HyperTransport, and Ethernet. The overhead associated with sending data over these types of connections requires a relatively large block of data to efficiently use the transfer bandwidth. Applications, such as signal processing, where a large block of data can be buffered and sent to the FPGA, can perform very well in these cases. However, other applications may not be able to buffer large amounts of data due to data dependencies.

Generally, the low cost and flexibility of reconfigurable hardware, along with the performance advantages of pipelining and parallelism, make reconfigurable hardware an attractive solution for applications requiring lower power or higher performance than a general-purpose

processor is able to provide.

2. Architectures Supporting Reconfigurable Accelerators

As FPGAs and other reconfigurable hardware have grown in popularity, researchers have been proposing architectures that integrate general-purpose processors with reconfigurable hardware to offer better interoperability between the devices. With these architectures, the processor and FPGA can operate on the same data and communicate with much lower overhead costs, allowing the FPGA to serve as a reconfigurable accelerator. One method of communication allows the processor to generate data or load data from memory and then send the data over a dedicated, low-overhead connection to the FPGA. One drawback of this method is that the processor spends time fetching the data and sending it to the FPGA, preventing it from performing other tasks [10]. However, applications that require the processor to operate on the data before or after the accelerator will benefit from this setup, because the processor and accelerator are very closely connected. The processor only has to fetch data from memory once and both the processor and accelerator can operate on it. A second method connects the FPGA directly to the memory system and allows it to make memory requests just like any other processor [10]. A third method allows the processor to set up direct memory access (DMA) transfers to the FPGA from memory [6]. The second and third methods allow the processor to perform other computation while data is sent to the FPGA.

This chapter explores some of the proposed and currently available designs that combine general-purpose processors with reconfigurable hardware, allowing for tightly-coupled computation. Section 2.1 describes some of the existing methods for processor and FPGA communication using external transfer buses. The description of the first architecture, the Triple Buffer Architecture, analyzed in the performance results is presented in this section. Section 2.2 explores the proposed and existing enhancements to general multi-core architectures for combining processors with reconfigurable hardware. Section 2.3 introduces the DMA Reconfigurable Accelerator Architecture (DRAA), which is the second architecture analyzed in the results. Section 2.4 presents the Data Pump Architecture (DPA), which is

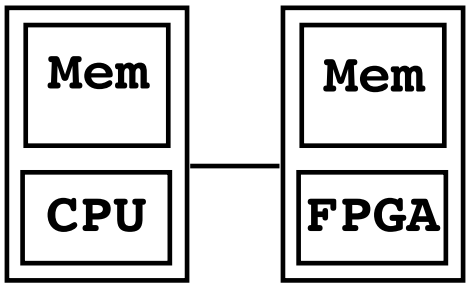

Figure 2.1: Common CPU-FPGA Architecture

the third architecture used in the performance results.

2.1 External Transfer Bus Architectures

One of the common, existing methods for combining general-purpose processors and FPGAs is to use an external communication bus. Both the processor and FPGA are separate, independently operating devices. Both devices have their own local memory for storing data. Figure 2.1 illustrates this setup. The connection between the two devices can be USB, Ethernet, HyperTransport, PCI-Express, or another communication protocol.

Many of the Xilinx Virtex FPGA devices [24] come packaged on a board that supports USB, Ethernet, and PCI-Express. The combination of the Matlab Simulink [16] and Xilinx System Generator [23] platform provides the ability to use USB or Ethernet to perform hardware co-simulation. The platform allows for designs to be partitioned between software simulation and hardware. The Xilinx FPGAs also support higher performance communication over the PCI-Express bus.

The company DRC Computer packaged a Xilinx Virtex 4 FPGA onto a board supporting communication with a processor over a HyperTransport bus [3]. Figure 2.2 shows a diagram of the DRC board and its connection to the processor. The FPGA has access to DDR memory and local low-latency RAM (LLRAM).

The drawback of these setups is that the overhead of preparing data to transfer and sending it over an external transfer bus can be significant. Unless the devices are sending a large enough block of data, such that the bus bandwidth is efficiently used and the

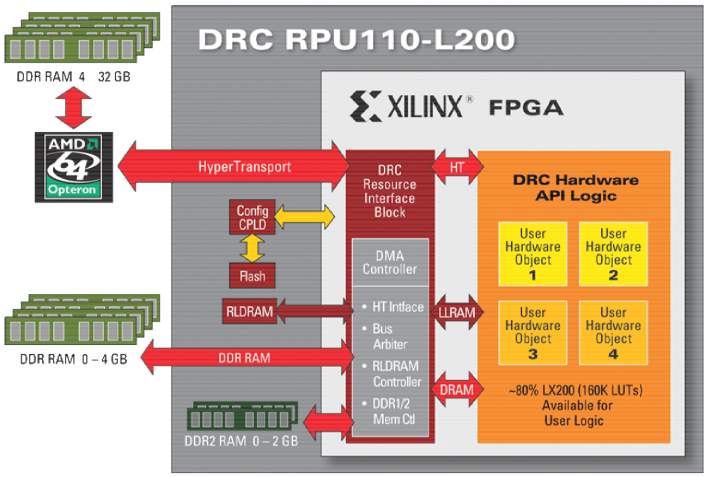

Figure 2.2: DRC Architecture

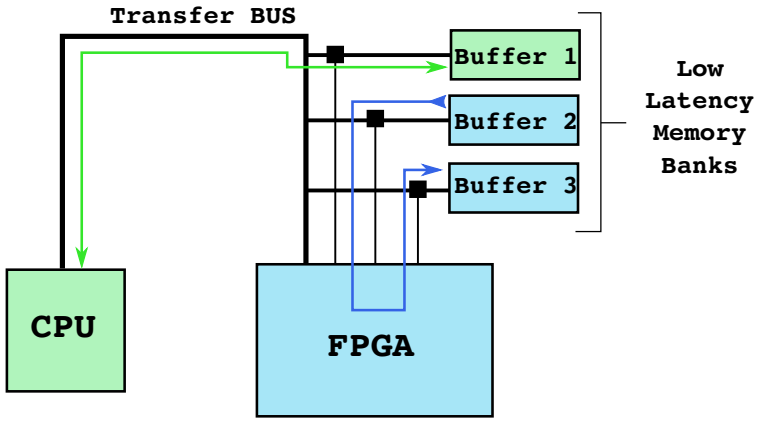

Figure 2.3: Triple Buffer Architecture

overhead of transferring is overshadowed by the transfer time, it is inefficient to send data to an external device. In addition, the bottleneck in the system is typically the rate at which the devices can send and receive data. For example, an FPGA with a single input port that is n bits wide, can only receive n bits of data per cycle, no matter how fast the transfer bus is able to send data. The slower clock rate of FPGAs limits the bandwidth of data in and out of the FPGA.

In an attempt to fully utilize the existing architectures to combine the processing abilities of processors and FPGAs, an extension to the setup shown in Figure 2.1 is presented. The

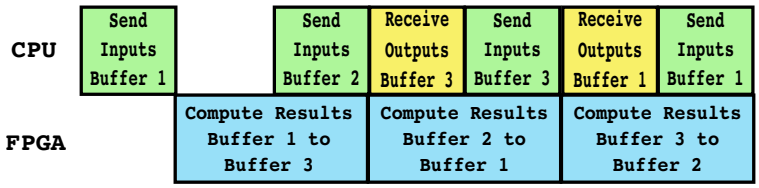

Figure 2.4: Streaming Schedule

Triple Buffer Architecture, shown in Figure 2.3, utilizes a simple buffering strategy to maintain a consistent stream of data through the hardware accelerator implemented on the FPGA. The processor prepares and sends a block of data across the transfer bus into a local buffer for the FPGA, taking advantage of the high bandwidth of the bus. While the processor is sending the next batch of data, the FPGA computes on the current data set by streaming it from one buffer, processing it, and streaming it into the remaining buffer. When both the FPGA and CPU have completed their tasks, the roles of the buffers switch. The processor uploads the output data from buffer and sends the new data into the buffer, while the FPGA processes the previously downloaded set of data.

The streaming schedule in Figure 2.4 shows how the data moves through the system over time. The CPU must initially send two blocks of data before it starts receiving outputs. Using this schedule there is no competition for the buffers between the CPU and FPGA or between reading and writing. This setup reduces conflicts in the data flow and allows for a nearly continuous stream of data through the FPGA.

The drawback of this method is that an application must be able to prepare blocks of data in advance. With some data dependent algorithms it can be difficult to prepare the next set of data if it is affected by the data currently being processed. In this case, it would be necessary for the FPGA accelerator to keep data that will be used on the next iteration, instead of returning it to the CPU. For applications with little data dependence and streams of data to process, this streaming setup can provide a strong advantage.

To investigate the performance of the Triple Buffer Architecture, a prototype was implemented using the DRC board with the Virtex 4 FPGA. The processor is able to send data to the FPGA over the HyperTransport bus. Because the DRC board does not have

enough independent memory banks to implement the three buffers, internal BlockRAM on the FPGA was used for the buffers. While this limits the performance of the prototype because the BlockRAM operates at the FPGA clock rate, it can provide an initial performance estimate.

2.2 General Multicore Architecture Enhancements

With the increasing popularity of multicore architectures, a great amount of time and effort has been spent in improving their performance. One of the increasingly popular improvements is to create systems with heterogeneous cores that take advantage of different technologies. One example is to combine general-purpose processing cores with reconfigurable hardware cores. Many different architectures combining these technologies have been proposed or already exist.

Garcia and Compton present one possible architectural design that incorporates a processor core and a reconfigurable core on the same chip [10]. Figure 2.5 shows a diagram of their proposed architecture. The design incorporates the processor and reconfigurable hardware as two cores in a multicore design. Each core has its own L1 cache and a shared L2 cache. The reconfigurable hardware has a memory management unit that is responsible for servicing memory requests from the hardware and performing virtual memory address translation. The reconfigurable core is able to read and write into the memory system through the memory management unit and is able to load or store an entire cache line at once. In addition, the processor can pass messages or buffers to the reconfigurable core through the memory management unit.

By interfacing with the existing memory system, the reconfigurable hardware can take advantage of many of the existing features of multicore design, including cache coherency and prefetch buffers [10]. Depending on the application, the use of cache coherency could provide either a performance increase or decrease for the system. If the processor and reconfigurable hardware both need to access the same sections of data, the overhead of the cache coherency protocol could reduce system performance.

Yan et al. propose a system that connects multiple reconfigurable processing units

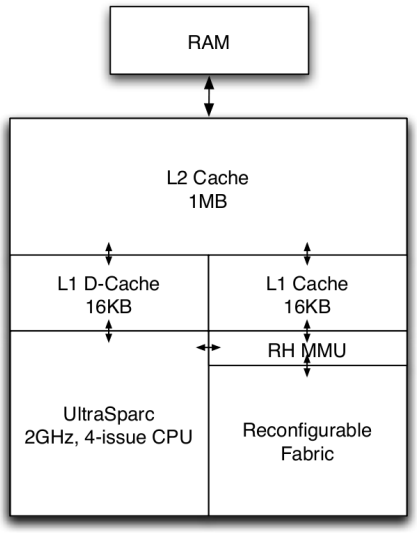

Figure 2.5: Reconfigurable Processor Architecture [10]

(RPU) to a multicore processor by a crossbar switch [28]. In this system, the processor cores access main memory and send data into local buffers for the reconfigurable units. A RPU Manager controls the reconfiguration and execution of the RPUs. One advantage of this design is that it allows for multiple RPUs that each have their own data input. While a system with one larger RPU could be designed to have multiple internal processors, the input bandwidth will be restricted by a single data port. Having multiple input ports allows for a greater bandwidth into the RPUs. Although this system puts the strain of supplying data onto the processors, it simplifies the design of the RPUs and does not expose the memory system to the RPUs [28].

Watkins and Albonesi introduce a design, called ReMAP, which allows multiple processing cores to share a reconfigurable fabric [19]. With this design there are multiple options for how the processing cores use the fabric. One option allows each core to have its own section of the reconfigurable fabric which it uses as its own accelerator. Another option allows one core to provide inputs to the reconfigurable fabric, while another core receives the outputs in a producer-consumer model. A third option allows all of the cores to share and synchronize on a single reconfigurable core. Figure 2.6 illustrates each of these options.

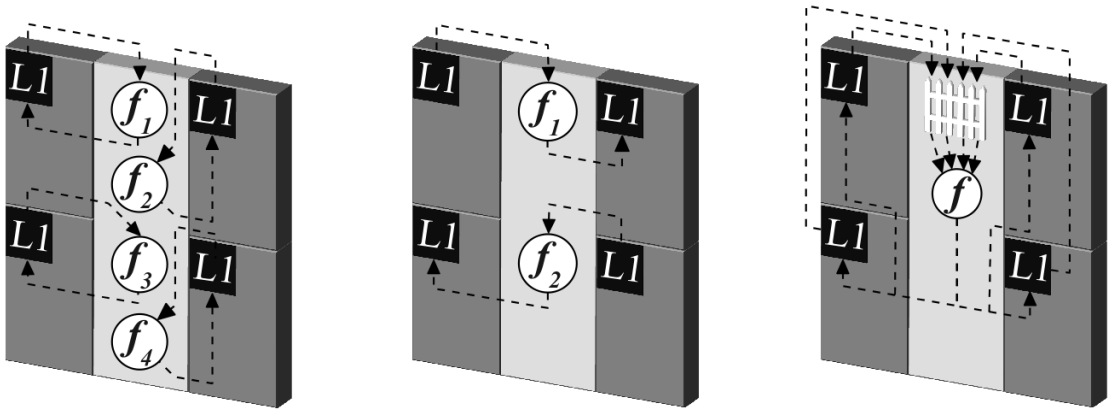

Figure 2.6: ReMAP Processing Configurations [19]

Xilinx FPGAs allow for reconfigurable hardware designs to incorporate processors. The Virtex-II Pro has Power PC cores available to execute software and interface directly with the FPGA [22]. More recent releases of the Virtex boards do not include the Power PC cores, but designs are able to use the Microblaze soft core in designs [27]. Although the Microblaze is not a dedicated hardware processor, it can be configured for varying area and performance restrictions. The designer can remove unwanted features to save area and increase performance. While the Power PC and Microblaze integrate very closely with the FPGA, the low performance of these processors makes them unsuitable for some high-performance applications that wish to combine processors and reconfigurable hardware.

Xilinx recently adopted the Advanced eXtensible Interface (AXI) protocol as its default communication protocol among Xilinx Intellectual Property (IP) cores [25]. The AXI protocol is part of a group of protocols, designed by ARM Limited, for embedded system design, called Advanced Microcontroller Bus Architecture (AMBA). This group provides communication protocols for different devices in an embedded system. The AXI protocol is useful for connecting processors and peripherals, such as accelerators, through a standard interface.

Intel has released information on a product that integrates an Intel Atom processor with an Altera FPGA in the same package[11]. This architecture provides a dedicated PCIe lane to communicate between the processor and FPGA. This connection allows the FPGA to

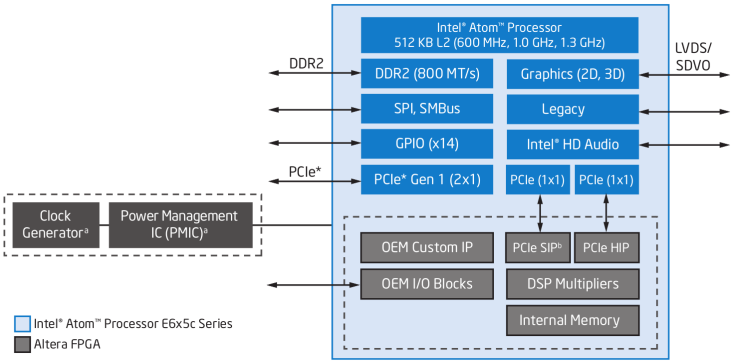

Figure 2.7: Intel Atom E6x5C Feature Diagram [11]

receive streams of data from the processor. Figure 2.7 shows a diagram of the architecture. While the performance of the Atom processor may not compete with many of the available high-performance processors, the Atom has low power consumption allowing for low power embedded designs.

2.3 Direct Memory Access Reconfigurable Accelerator Architecture

The Direct Memory Access (DMA) Reconfigurable Accelerator Architecture (DRAA) [6] combines a general-purpose processor with a reconfigurable hardware accelerator over a local bus. A DMA module is responsible for providing a stream of data from memory to the hardware core and storing the core's output data back to memory. The processor sets up transfers and sends them to the DMA module to execute. The processor has control over what input data goes to the accelerator and where to store the output data in memory, giving the processor control over the execution process.

Figure 2.8 shows a diagram of the DRAA architecture. This architecture is another extension to the general multicore architecture and is similar to many of the architectures in Section 2.2.

Section 2.3.1 describes the AXI protocols, which provide the interconnection between components in the system. Section 2.3.2 explores the details of the Xilinx AXI DMA module. Finally, Section 2.3.3 describes the prototype implementation of the DRAA.

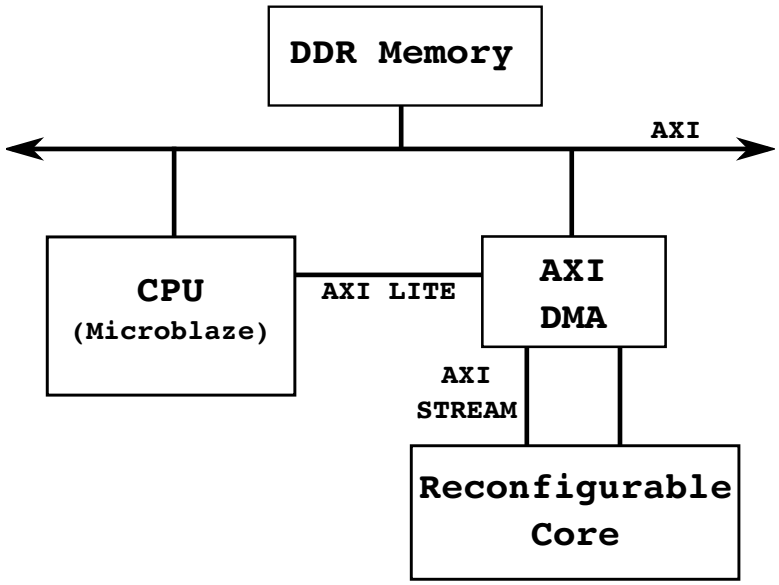

Figure 2.8: Direct Memory Access (DMA) Reconfigurable Accelerator Architecture (DRAA) [6]

2.3.1 Advanced Microcontroller Bus Architecture

The Advanced Microcontroller Bus Architecture (AMBA), designed by ARM Ltd, provides a group of protocols for high-performance interconnection between processors and other system components [15]. The protocols are designed for flexible, high-bandwidth and low-latency communication for a wide range of components. The Advanced eXtensible Interface (AXI), one protocol from the AMBA group, has separate read and write channels that support burst transfers. In addition, the AXI-Lite [15] and AXI-Stream [14] protocols allow for simpler communication with less complexity than the AXI protocol.

AXI

The AXI protocol defines Master and Slave interfaces which define how data moves between components. The Master has control over the flow of data, allowing it to write data to the Slave and read data from the Slave. A channel provides a connection between a Master and Slave allowing for the transfer of information. Each channel has a VALID and READY signal to coordinate handshaking between the Master and Slave. When the

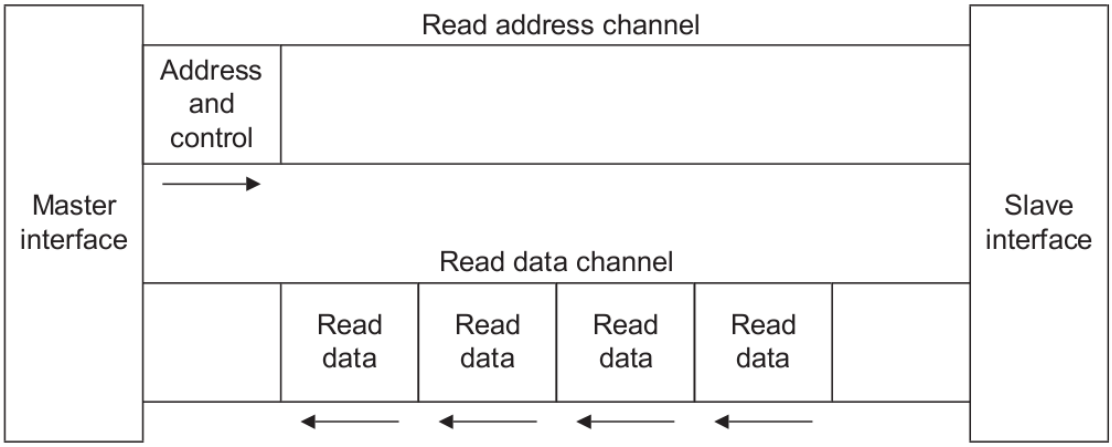

Figure 2.9: AXI Read Channels [15]

READY signal is high, the receiving component is indicating that it is ready to accept a transfer. The VALID signal is set high to indicate that there is valid data on the channel. The LAST signal is also included for data channels to signal the end of a transaction. With the use of the LAST signal, multiple addresses can be sent back-to-back. The LAST signal can then be used to indicate the breaks in the data between one address and the next [15].

To initiate a read, the Master sends address and control information on the read address channel to the Slave. The Slave returns the requested data on the read data channel. The read data channel can be between 8 and 1024 bits wide, in powers of 2. Figure 2.9 illustrates the read transaction between the Master and Slave [15].

The Master initiates a write by sending address and control information on the write address channel while sending data on the write data channel. The write data channel can be between 8 and 1024 bits wide, in powers of 2. The write response channel allows the Slave to acknowledge the receipt of a burst of write data. Figure 2.10 illustrates the write transaction between the Master and Slave. The Master can initiate new write transfers without waiting for acknowledgement of previous transfers, because it is assumed that the Slave will buffer all received data [15].

The AXI protocol supports multiple Masters and multiple Slaves connected through an interconnect, while keeping the standard Master and Slave interfaces for connection of each

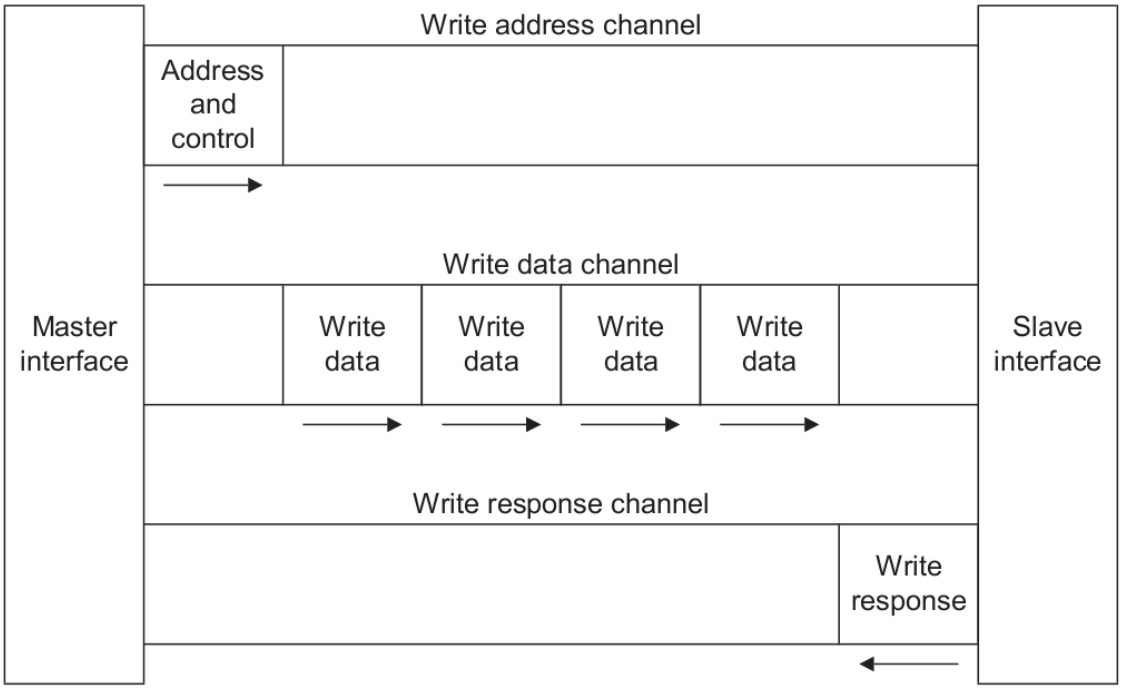

Figure 2.10: AXI Write Channels [15]

component to the interconnect. AXI supports multiple interconnect strategies, depending on the desired performance of the system, including sharing the address and data buses between components, sharing the address bus and having multiple data buses, or having multiple address and data buses. To facilitate sharing of the channels, each transaction on the AXI protocol is assigned an ID tag. While data within a transaction must be sent in-order, different transactions can be sent out-of-order, using the ID tag as an identifier [15].

AXI-Lite

The AXI-Lite interface contains a subset of the AXI functionality and is intended for sending control signals between components. AXI-Lite does not support burst transfers and only allows data accesses that are the same width as the data bus. The data bus can be either 32 bits or 64 bits wide in AXI-Lite. Transaction IDs are not used in AXI-Lite meaning that all transfers must happen in-order [15].

AXI-Stream

The AXI-Stream interface is designed to transmit streams of data, without addressing, between components. It supports both single Master to Slave connections and connection between multiple Masters and Slaves. The protocol also supports optional Control/Status streams that deliver additional transfer information in parallel with the main data channel [14].

Data is grouped into packets to be sent across the AXI-Stream interface. A single packet may have to be broken into multiple transfers, each identified by a VALID-READY handshake, across the connection. A frame can be used to represent a group of packets that are processed together [14].

The protocol supports the transmission of data bytes, position bytes, and null bytes. A data byte contains valid data in a stream. The position byte does not contain valid data, but serves as a placeholder in the stream to maintain the relative positions of other bytes. A null byte does not contain valid data and does not affect the position of bytes within the stream [14].

Through the use of the different byte types, the protocol supports different types of data streams. The byte stream consists of data bytes and null bytes. Null bytes can be inserted anywhere within the stream without affecting the data being sent. A continuous aligned stream only sends data packets and all of the data is assumed to be aligned in the correct positions. A continuous unaligned stream sends a sequence of data bytes, but position bytes can be used at the beginning or end of the group to shift the position of the bytes. Lastly, a sparse stream allows data and position bytes to be mixed within a stream of data [14].

2.3.2 AXI DMA

The Xilinx AXI Direct Memory Access (AXI DMA) [26] core is designed to provide high-bandwidth direct memory access between memory and a peripheral device. The AXI DMA core connects to the AXI bus, which is also connected to the memory, to handle memory access. The peripheral device is connected to the AXI DMA core by the AXI-

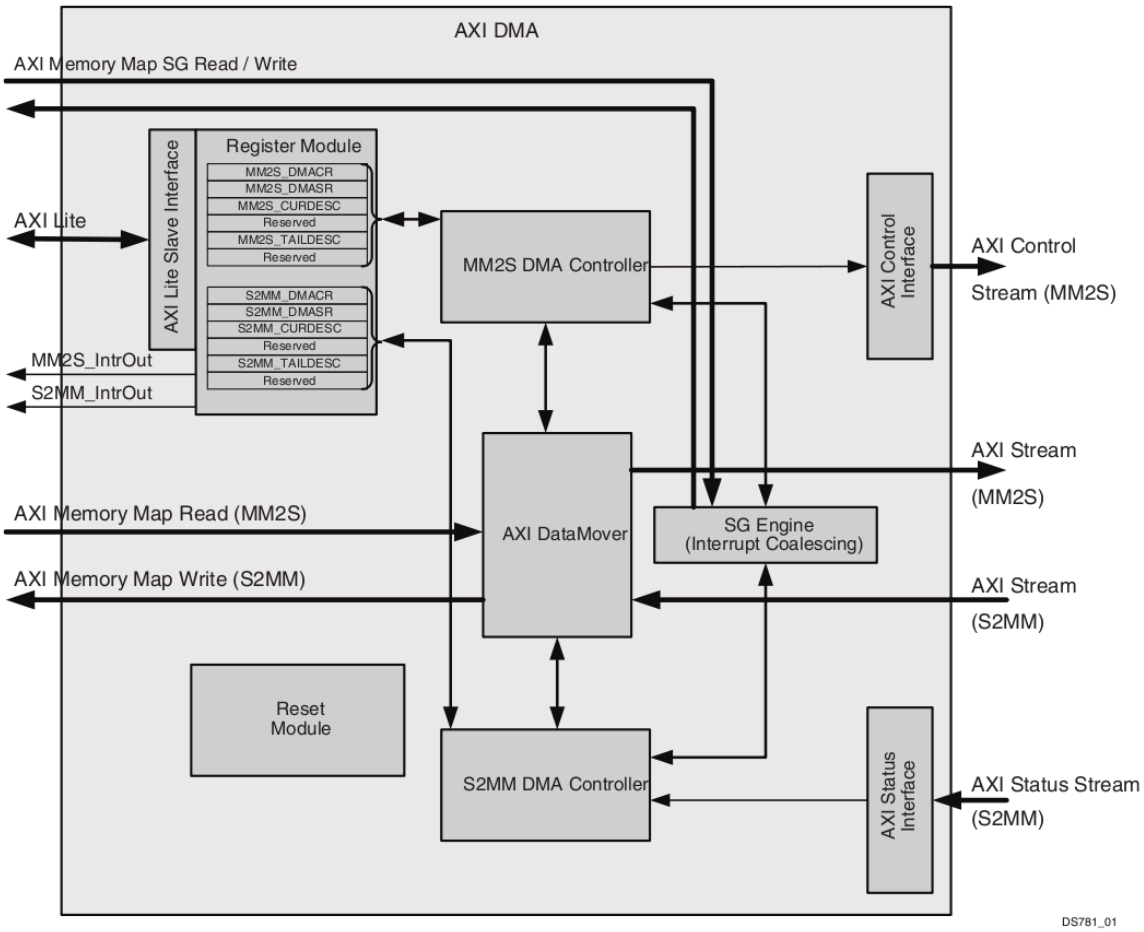

Figure 2.11: AXI DMA Block Diagram [26]

Stream protocol with two independent connections. One connection sends data from the DMA module to the peripheral, while the other connection allows the peripheral to send data to the DMA module. An AXI-Lite interface is used to connect the DMA module to a processor, so that the processor can setup transfers for the DMA module to execute. The DMA module allows the processor to offload the data fetching so that it can perform other tasks while the transfers complete.

Figure 2.11 shows the diagram of the AXI DMA module. The Data Mover is responsible for transferring data from the memory over the AXI Memory Map Read channel and sending the data to the peripheral over the AXI Stream MM2S channel. The Data Mover also receives streams of data from the peripheral over the AXI Stream S2MM channel and stores

the data to the memory over the AXI Memory Map Write S2MM channel. The Scatter Gather Engine coordinates the execution of the DMA requests and allows for the collection of data from different locations in memory to form a single block of data. The optional Control/Status interface allows the DMA module and peripheral to communicate extra information about the transfer. The DMA module is able to interrupt the processor when transfers are complete or other events occur through the MM2S_IntrOut and S2MM_IntrOut connections [26].

The AXI DMA module also supports asynchronous clock domains for the different AXI connections. Having asynchronous clock domains allows the DMA module to retrieve data from memory at one clock frequency, but feed the streams of data to a slower clocked peripheral through the stream connections. Lastly, the AXI-Lite interface for setting up transfers can operate at its own clock frequency, different from the AXI and AXI-Stream interfaces [26].

Information about the desired DMA transfers are stored in a data structure, called a Buffer Descriptor (BD), by the processor. Each BD holds the information about the data's location in memory and length in bytes. The BDs are held in a ring where each BD has a pointer to the next one in the ring. This structure allows the processor to set up a group of transfers to be processed in sequence. By using the Start of Frame bit (TXSOF) and End of Frame bit (TXEOF), BDs can be grouped into packets. At the simplest level, each BD can set both the TXSOF and TXEOF flags, indicating that its block of data represents an entire packet. However, the Scatter Gather Engine supports packets composed of data from different locations in memory. In this case, the first BD in the packet sets the TXSOF bit and the last BD in the packet sets the TXEOF bit. There is a second ring of BDs to indicate where the data returning from the peripheral should be stored in memory. The DMA module sets the RXSOF and RXEOF bits in the BDs as it receives data from the peripheral [26].

The DMA module automatically processes through a ring of BDs until it reaches the pointer to the last BD in the ring, called the TAIL pointer. As it processes each BD, the DMA module updates the information in the BD to indicate its status to the software. Once

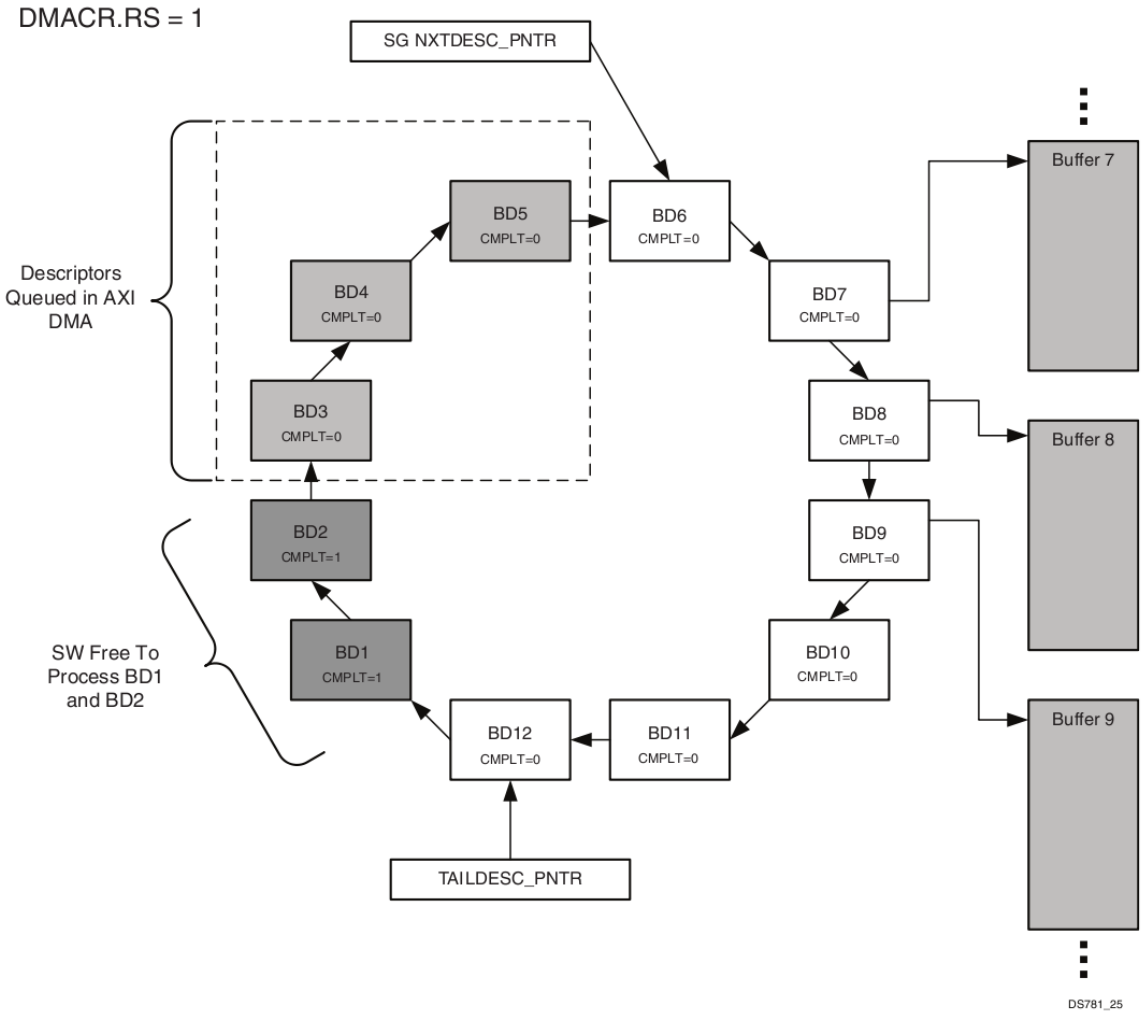

Figure 2.12: DMA Execution on the Buffer Descriptor Ring [26]

a BD has been processed and the Complete bit has been set, the processor can fetch the BD and reallocate it for a new DMA task. Therefore, it is feasible for the processor to keep the DMA constantly busy by setting up some initial DMA requests and then continuing to setup new DMA requests as the DMA module processes the original tasks [26].

Figure 2.12 illustrates the ring of BDs at a point in the middle of execution. The first two BDs have been processed and have their complete bit set. BDs 3,4, and 5 are currently queued in the DMA module for processing. BDs 6-12 are waiting to be processed by the DMA module. The TAIL pointer currently points to BD12 indicating that it is the last BD

in the ring to be processed [26].

2.3.3 Architecture Implementation

A prototype model of the DRAA was implemented on a Virtex 6 FPGA [24] using the Xilinx EDK tool [20]. This model uses a Microblaze processor to represent the general purpose-processor in the design with DDR3 RAM as main memory. AXI creates the local bus connecting the processor, DDR3 memory, and DMA module. The Xilinx AXI DMA unit connects to the bus and supplies the reconfigurable hardware with data using the AXI-Stream protocol. For the current implementation, the AXI bus and AXI-Stream connections are 64 bits wide. An AXI-Lite bus connects the processor and AXI DMA module so that the processor can setup transfers to and from the hardware. Lastly, a hardware timer, under the control of the processor, measures the execution cycles of the system.

To provide an accurate performance model of the DRAA, the system was configured to run the processor, AXI bus, and DMA module at 100MHz, while the reconfigurable hardware core runs at 10MHz, consistent with a 10 times ratio between processor and reconfigurable hardware clock frequencies. The implemented architecture, with a Microblaze processor, serves as a prototype model for an architecture with a modern processor connected to a reconfigurable accelerator.

2.4 Data Pump Architecture

The Data Pump Architecture (DPA) [17], designed as a signal processing platform, allows the user to control the data movement throughout the system so that data can be efficiently delivered to the vector processing units. The system is part of the SPIRAL project and was developed by researchers at Carnegie-Mellon University, University of Illinois at Urbana-Champaign, and Drexel University. The system does not utilize data caching, but expects the user to maintain the necessary data in local memory. The system is also configurable so that users can change the architecture configuration to fit their application. Figure 2.13 shows a diagram of a basic configuration of the DPA. External memory is an off-chip high-capacity bank of memory, while local memory is a smaller on-chip bank

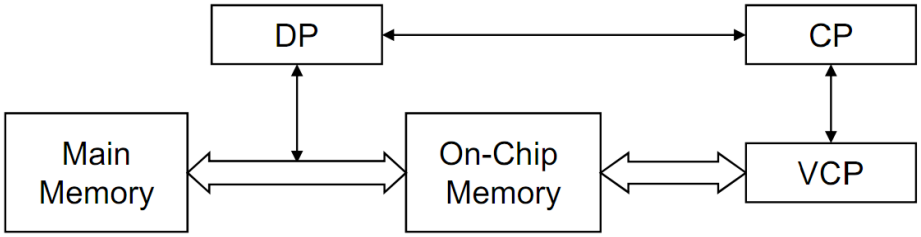

Figure 2.13: Basic Data Pump Architecture Diagram [17]

of memory. External memory has a higher access latency than local memory. The Data Processor (DP) controls the movement of data between external memory and local memory. The Compute Processor (CP) moves data between local memory and the vector register file, where the vector units process the data.

The DP and CP processors support the SPARC V8 ISA and the extensions provided by the DPA ISA [17]. The processors are single-issue and may issue one SPARC or one DPA instruction per cycle. Typically, the DP and CP do not operate on data, but just control its movement to the vector processors. With data-dependent applications, it is possible for the DP and CP to access data located in local memory.

In more advanced configurations, the DPA can have multiple CPs and split the local memory into segments. Each CP is given read and write permissions for each of the local memory segments. A mailbox allows the CP and DP to communicate and synchronize the execution process. In addition, counters on the local memory segments allow the DP and CP to synchronize by waiting for a specified number of reads or writes to a local memory segment. Figure 2.14 shows a more detailed diagram of the DP processor and associated architecture. Figure 2.15 is a more detailed diagram of the CP and vector processing units [17].

The DPA architecture has been implemented in a VHDL and Verilog design. The design can be simulated using RTL simulators or implemented on an FPGA device for testing and verification of designs. A simulator and performance model exist for early stage testing and development and exploration of the DPA architecture [13].

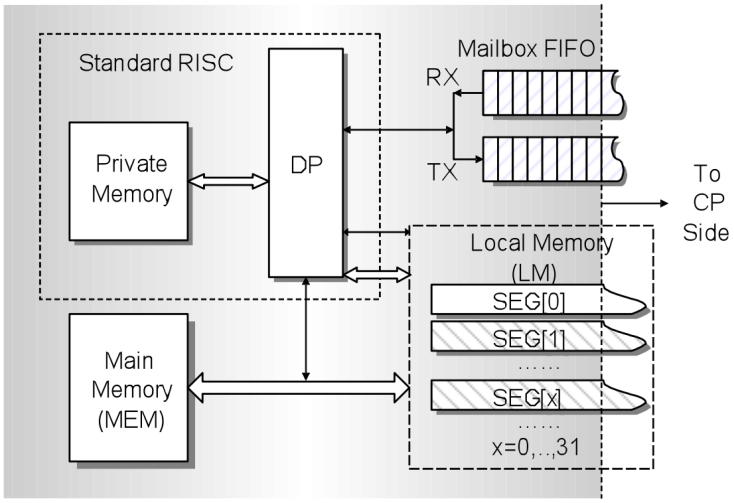

Figure 2.14: DP Architecture Diagram [17]

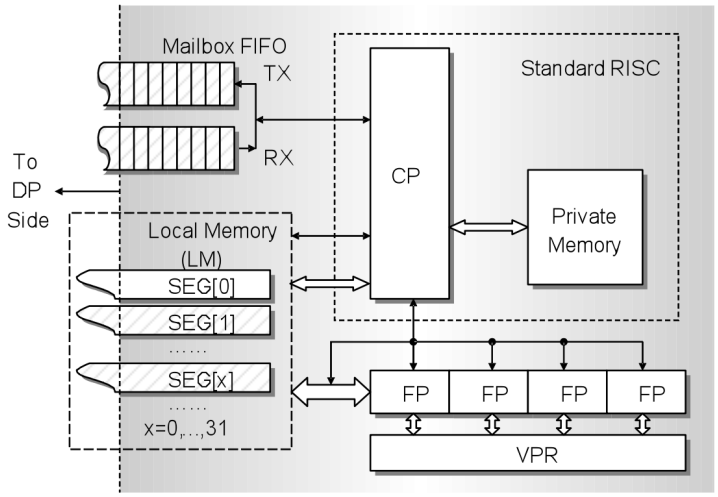

Figure 2.15: CP Architecture Diagram [17]

While the DPA is intended for signal processing applications, the memory movement architecture is also suitable for other applications that process large amounts of data. By replacing the standard vector processing units with a custom accelerator, the DPA can be used to efficiently manage the data sent to the accelerator.

2.4.1 DPA Simulator

The DPA Simulator [13] provides performance modeling and simulation of designs on the DPA architecture. It allows users to explore explore the potential performance gain from using the architecture without having to implement hardware designs. It also provides a platform to explore modifications to the DPA architecture.

The simulator uses threads to simulate the different processors running in the DPA architecture. Instructions that match the DPA ISA [17] are provided so that code written for the simulator can be translated to run on an implementation of the DPA architecture. The DPA ISA specifies instructions for moving data between external and local memory, moving data between local memory and the vector registers, performing vector operations, and synchronization through counters or mailboxes. The simulator focuses on modeling the memory transfer and vector computations, but does not model the actual processor cores [13].

As the simulator executes, it generates a trace of instructions which are analyzed by the performance model at the end of execution. The performance model uses the information from the trace to synchronize the executions of the different processors and generate cycle counts for each operation. A final report that provides the number of execution cycles for each processor and the overall execution time is generated by the performance model.

3. Merge Hardware

The main operation in the Gaussian LU algorithm is the sparse row addition (merge) of rows to eliminate elements below the diagonal. Since data-dependent branching and row element indexing overhead reduce the performance of sparse row addition on general-purpose processors, a custom hardware accelerator can be used to perform the row merging and increase LU performance. Utilizing reconfigurable hardware to design an accelerator allows for a low cost and flexible implementation.

Section 3.1 introduces the design of the merge accelerator. Section 3.2 explains the theoretical performance of the accelerator. Section 3.3 introduces the supporting hardware design that manages multiple rows and their distribution across multiple merge units. Section 3.4 describes the details of including the merge accelerator on the proposed architectures.

3.1 Merge Hardware Design

A custom hardware core was designed to perform the addition of two sparse rows, u and v. The merge process compares the column numbers of the row elements puts them into a single output row in the correct order, adding the values of any two elements with the same column number. Figure 3.1 shows a diagram of the merge hardware design.

The merge core accepts two input rows, sorted by column number, and stores them in BlockRAM (BRAM) FIFOs. The core reads the input rows from the FIFOs and merges the row elements into a single output row. In order to maintain a consistent output data rate, a new input element must be read from the input FIFO each time that an element for that row moves into the output row. Because the FIFOs have a read latency of one cycle, the merge process must look one step ahead to generate the correct read requests. The merge hardware has two stages of registers so that is can successfully determine which element to replace in the next cycle.

On each cycle, one element moves into the output row. A floating point adder is used to add the values of elements with the same column number. A shift register keeps track

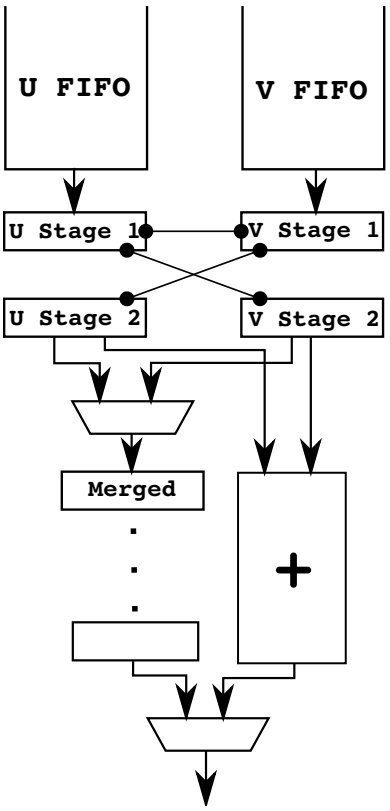

Figure 3.1: Merge Accelerator Design [5]

Table 3.1: Merge States and Transitions

Current State	Next Compare	Transition
$u_2 == v_2$	$u_1 == v_1$	Read u and v inputs
	$u_1 < v_1$	Read u input
	$u_1 > v_1$	Read v input
$u_2 < v_2$	$u_1 == v_2$	Read u and v inputs
	$u_1 < v_2$	Read u input
	$u_1 > v_2$	Read v input
$u_2 > v_2$	$u_2 == v_1$	Read u and v inputs
	$u_2 < v_1$	Read u input
	$u_2 > v_1$	Read v input

of the column number and unadded elements values for the latency of the adder. A second shift register tracks whether the added or unadded value will be used in the output row and a final multiplexer selects the correct value.

A total of six comparisons are performed in parallel during each cycle. The first three comparisons check for equality between the column numbers, while the other three comparisons check if one column number is less than another. The three column number pairs that are checked on each cycle are U Stage 1 and V Stage 1, U Stage 2 and V Stage 1, and U Stage 1 and V Stage 2. Table 3.1 shows the possible states and transitions for the merge core.

The first column shows the current state based on the two elements that are currently in Stage 2. If the two elements have equal column numbers, then both elements will merge into one element and move into the output row, allowing for both of the Stage 1 elements to move to Stage 2. This means that the look ahead comparison must compare the column numbers for both of the Stage 1 elements, as shown in Column 2, and issue the correct read requests, shown in Column 3. In the other two current states, only one of the Stage 2 elements moves into the output row, while the other element stays in place. In these cases, the Stage 1 element that will move into Stage 2 to replace the output element, must be compared with the element that stays in Stage 2.

3.2 Merge Performance

The merge core has an output data rate equal to the hardware clock rate. In other words, it is able to output one row element on each clock cycle. Because the core does not allow any gaps in the pipeline, the data rate remains constant for a pair of rows. The setup time between pairs of rows will introduce some gaps in the output, but the overall data rate will approach the clock rate for a consistent stream of data.

The latency of the merge core is dependent on the structure of the two input rows. The minimum possible latency is when the column number of every element in one input row is also present in the other input row, causing all of the elements in one row to be merged together. The minimum latency, shown in Eq. 3.1, will be the sum of the number of elements in the longer input row, the latency of the floating point adder, the latency of the input FIFOs, and two cycles for the compare stages.

$$L_{MIN} = max(NNZ(u), NNZ(v)) + L_{ADD} + L_{FIFO} + 2 \tag{3.1}$$

The maximum latency occurs when no two elements in the input rows have the same column number, meaning that every element in the input will be in the output. The maximum latency, shown in Eq. 3.2, is the sum of the number of non zeros in each input row, the floating point adder latency, the input FIFO latency, and two cycles for the compare stages.

$$L_{MAX} = NNZ(u) + NNZ(v) + L_{ADD} + L_{FIFO} + 2 \tag{3.2}$$

3.3 Merge Core Manager

To efficiently use the merge accelerator and see a benefit in execution speedup, the processor can set up a block of rows to be processed at once. During each LU iteration, a pivot row is selected and must be added to each of the other affected rows in the matrix. Sending all of these rows at once can allow for better utilization of the transfer bandwidth and merge accelerator.

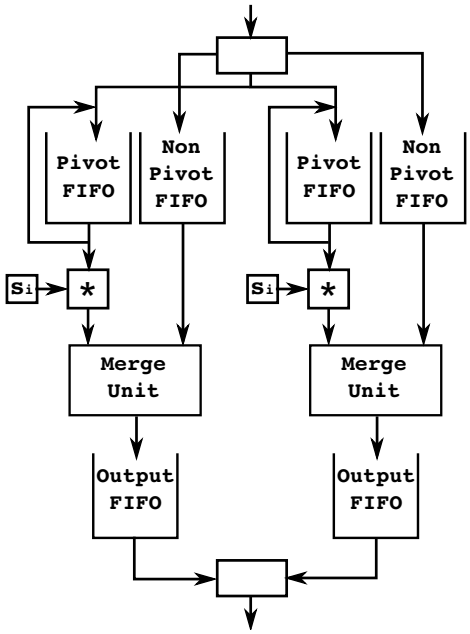

Figure 3.2: Merge Manager Design [6]

The pivot row is sent first, followed by each of the additional rows to be merged with the pivot row. When the pivot row arrives, it is stored in a separate BRAM FIFO. A second BRAM FIFO stores the elements from all of the non-pivot rows. An additional bit is stored with the rows to indicate the last element in each row. The AXI-Stream protocol provides the LAST signal so that the boundaries between rows can be determined in a stream of data.

Once data is ready in both the pivot and non-pivot FIFOs, the elements of the pivot row and one non-pivot row are streamed into the merge unit input FIFOs. Because the pivot row needs to be scaled, it is passed through a pipelined floating-point multiplier as it loads into the merge unit. The pivot row elements are recycled into the FIFO so they can be used with the next non-pivot row. After the last output element leaves the merge unit, the pivot row and next non-pivot row are loaded into the merge unit. The output elements are sent back to the DMA module over the output connection. Figure 3.2 shows a diagram of the merge core manager.

Because only one pair of rows can be in the merge unit at a time, the latency of processing

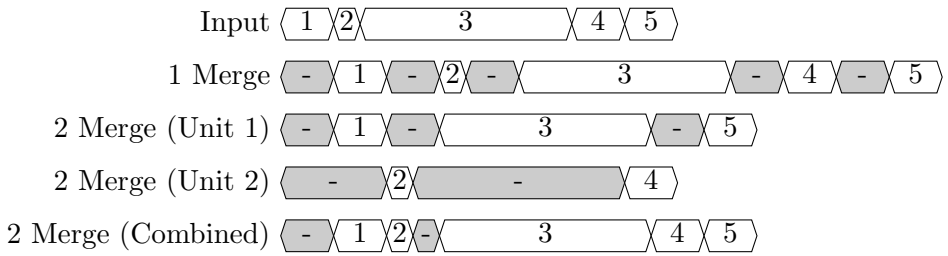

Figure 3.3: One vs Two Merge Units

rows in the merge unit causes gaps in the output data stream. To reduce these gaps, it is possible to have multiple merge units. The merge unit, pivot FIFO, and non-pivot FIFO must be duplicated for each merge unit. In addition, each merge unit must now have an output FIFO to hold data while it waits to be sent on the outgoing connection. As row data arrives, the pivot row is loaded into the pivot row FIFO for all of the merge units. The non-pivot row data is distributed among the different merge cores as it arrives. The manager alternates the outgoing connection between the output FIFOs after each full row. The manager pulls an entire row from an output FIFO before moving to the next output FIFO so that the rows are output in the same order they arrived and get sent to the correct location in memory.

The latency of the merge unit introduces a gap after each row. This means that one merge unit will not fully utilize the output port, as shown in Figure 3.3. Having a second parallel merge unit allows the hardware to almost fully utilize the output port. Some gaps may still exist if there are short rows, less than the latency of the merge unit. Having more than two merge units will help to fill in these additional gaps, but will only provide a limited performance benefit, because the output port becomes saturated.

Figure 3.3 demonstrates the activity on the output port with one and two merge units. The activity shows the row numbers 1, 2, 3, 4, and 5 being merged as time progresses. The blanks indicate idle activities. The top row shows the arrival of the row data on the input port. With a single merge unit, there is a gap after each row. For two merge units, each unit is shown individually as the input rows are alternated between them. The combined

output alternates between the outputs of the two merge units, showing the utilization of the output port. The short row introduces a small gap, because the other merge unit does not have outputs available yet.

3.4 Merge Accelerator Architecture Integration

3.4.1 Merge Accelerator in Triple Buffer Architecture

To integrate the merge accelerator with the Triple Buffer Architecture, a module that reads input data from memory and supplies it to the merge core and writes output data back to memory is necessary. The module synchronizes with the CPU to rotate the buffers and then begins streaming data from the input buffer. When the merge accelerator begins providing output data, the module streams data to the output buffer. When the processor has finished uploading the results and downloading new inputs, it sends a synchronization signal to the accelerator. When the accelerator has finished processing the current batch of rows, it responds to the synchronization signal and begins processing the next batch of inputs.

The processor has control over the execution process by sending input data to the accelerator and reading output data from the accelerator. The processor is also responsible for sending the synchronization signal to the accelerator.

3.4.2 Merge Accelerator in DRAA

The Merge Core Manager integrates into the DRAA by connecting to the Xilinx AXI DMA module. The DMA provides streams of data to the merge accelerator using the AXI-Stream protocol. The merge manager is able to receive the stream of data and direct it to the correct input FIFOs. When the merge accelerator outputs data, the manager sends the data on the outgoing AXI-Stream connection to the AXI DMA module.

The processor sets up blocks of rows to be merged by sending the transfer information to the AXI DMA module. The module fetches the rows and sends them through the merge accelerator. When a pair of rows has been merged, the AXI DMA sends the data back

to memory. As with the Triple Buffer Architecture, the processor has control over the execution process. It determines what data goes to the accelerator by setting up the DMA transfers. The processor can also decide to not use the accelerator and perform the merging process itself. This can be beneficial for small rows that will not benefit from being sent to the accelerator, due to the associated overhead.

3.4.3 Merge Accelerator in DPA

Using the DPA simulator, the benefit of using the merge accelerator on the DPA architecture to perform sparse LU was investigated. The simulator was modified to add an instruction for the merge operation. Performance modeling information for the merge accelerator hardware was added to the performance model so that the performance results can accurately account for the execution time of the merge. To separate the loading of the pivot row from the non-pivot rows, an instruction that loads the pivot row into the merge unit was added. The merge instruction handles the transfer of the non-pivot rows and the output row. Adding the support for the merge instruction in the simulator and performance model were the only modifications needed to support sparse LU. Although it is not typical in the DPA, the CP and DP must also access data in local memory since sparse LU is a data-dependent algorithm.

Algorithms 3 and 4 show the basic flow of the sparse LU algorithm on the DPA. Figure 3.4 depicts these algorithms using flow charts. The DP loads the submatrix rows from external memory into local memory. The CP performs the merging process on the submatrix rows and writes them back to local memory. The CP also updates the column map with the fill-in information generated by the merge process. The two processors synchronize using the local memory counters. These counters provide a count of the number of reads or writes to a local memory segment.

To improve performance and support larger matrices, the basic algorithms were improved. For larger matrices, all of the submatrix rows may not fit in local memory at once, so the submatrix is broken into smaller blocks and each block is processed. To improve the throughput of the system, a double buffering strategy is used. After loading one block of

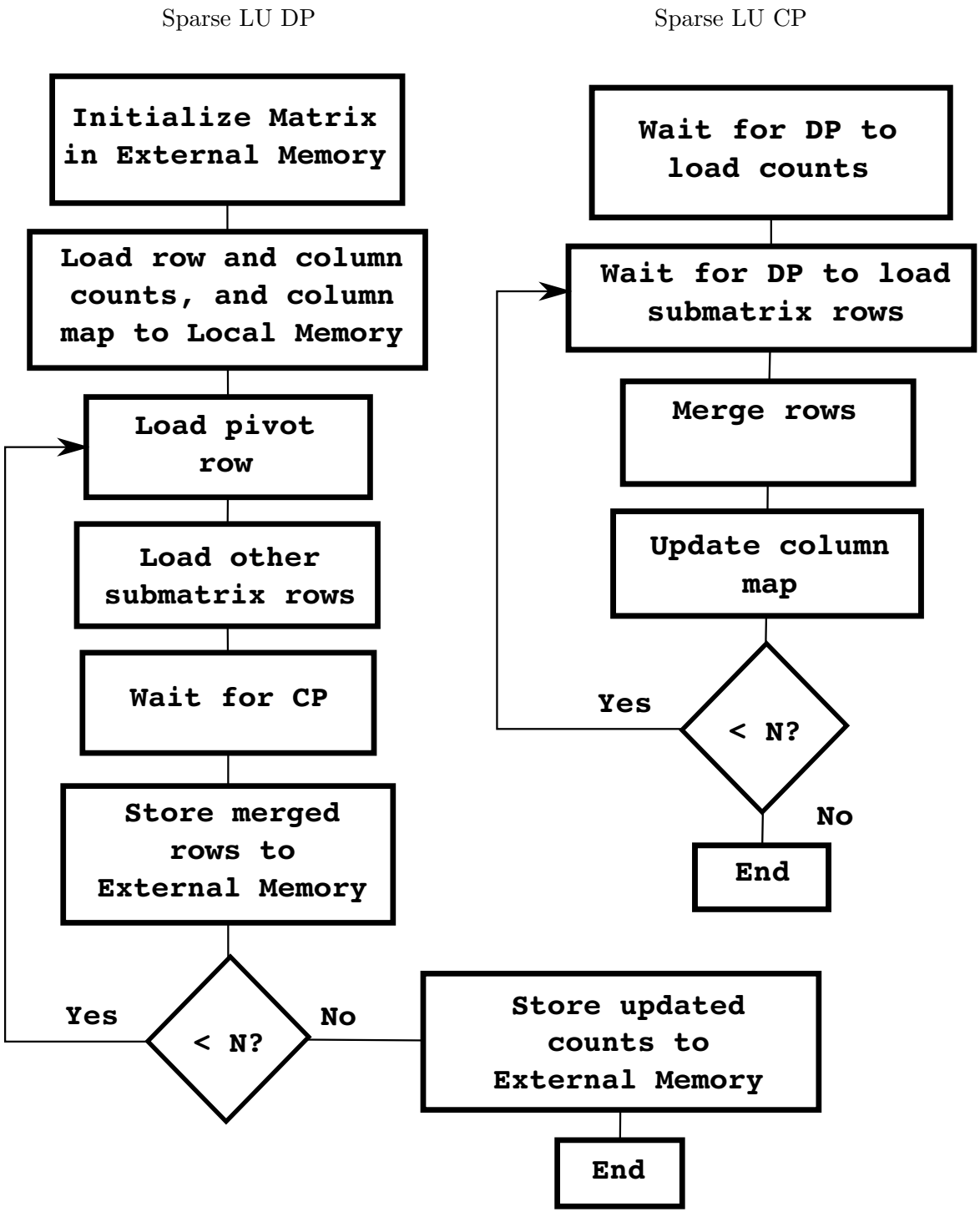

Figure 3.4: DPA Sparse LU Flow Diagrams

Algorithm 3 Sparse LU Basic DP Algorithm

Initialize Matrix in External Memory
Load row counts, columncounts, and column map in Local Memory
for $i = 1 \rightarrow N$ **do**
  Load Submatrix Rows into Local Memory
  Wait for CP to Write Output Rows and Column Map Updates
  Store Merged Rows to External Memory
end for
Store updated counts back to External Memory

Algorithm 4 Sparse LU Basic CP Algorithm

Wait for DP to Load Counts and Column Map
for $i = 1 \rightarrow N$ **do**
  Wait for DP to Load Submatrix Rows
  Merge Rows
  Update Column Map
end for

submatrix rows into an input buffer, the DP can prepare a second input buffer with the next block of rows. The CP alternates between processing the two input buffers. This strategy allows each processor to continue operating instead of idling while it waits for the other processor to complete its operations. This allows the system to process data more efficiently.

4. LU Software Analysis

This chapter analyzes the software performance of Sparse LU. The following chapters compare the performance of the accelerator architectures to the software performance. Section 4.1 analyzes the power system matrices, Section 4.2 analyzes random matrices, and Section 4.3 analyzes matrices from the UFSparse collection.

4.1 Power System Matrices

As described in Section 1.2, the Gaussian LU algorithm outperforms UMFPACK on power system matrices. Analysis of the Gaussian LU software performance provides a comparison benchmark for the accelerator architectures.

The sparse row addition (merge) function accounts for a large percentage of the execution time for power system matrices. By profiling the Gaussian LU software, the percent of execution time spent in the merge function can be measured. Table 4.1 shows the percentages for the different power benchmark systems. The merge function accounts for around 55%-65% of the execution time, depending on the system. The maximum LU speedup that can be achieved by speeding up the merge function can be calculated using Amdahl's Law, shown in (4.1).

$$Speedup = \frac{1}{(1-p) + \frac{p}{s}}, \tag{4.1}$$

Table 4.1: Gaussian LU Merge Profiling

System	% Merge	Maximum LU Speedup for 0-Execution-Time Merge
1648 Bus	65.6%	2.91X
7917 Bus	54.3%	2.19X
10279 Bus	55.8%	2.26X
26829 Bus	62.7%	2.68X

Table 4.2: Power Matrix Characteristics

System	Avg. NNZ per Row (Original)	Avg. Num Submatrix Rows	Avg. NNZ per Submatrix Row (During LU)
1648 Bus	7.1	8.0	18.9
7917 Bus	7.3	8.5	24.4
10279 Bus	7.0	8.3	31.0
26829 Bus	7.0	8.7	43.7

where p is the fraction of execution that is being improved and s is the speedup on that fraction.

Assuming an infinite speedup for the merge function, causing it to take zero execution time, the remaining fraction of execution limits the speedup. Column 3 of Table 4.1 shows the maximum LU speedup for each of the power benchmark systems.

Further analysis of the power system matrices during LU execution reveals information about the merges being performed. During each iteration, the submatrix is selected by finding all of the rows with their leading element in the current column. Table 4.2 lists the average number of non-zeros (NNZ) for all rows in the original matrix, the average number of submatrix rows selected during each iteration, and the average number of non-zeros in the group of submatrix rows.

All of the matrices start with around the same number of non-zeros per row, on average. As the LU execution proceeds, fill-in causes the rows to accumulate more non-zeros. As the system size grows, the average number of non-zeros per submatrix row increases.

The number of submatrix rows selected on each iteration is about the same for each system, on average, but the number of rows grows slightly as the system size increases. These results confirm that the groups of rows being merged are relatively small, but the groups become larger as the system size increases.

Figure 4.1 shows the distribution of NNZ per row in the original matrix, while Figure 4.2 shows the distribution of the number of rows per submatrix for the 10279-bus power system. As shown in Table 4.2, the averages for both the NNZ per row and the number of rows in the submatrix are low. The distributions confirm that the majority of submatrices

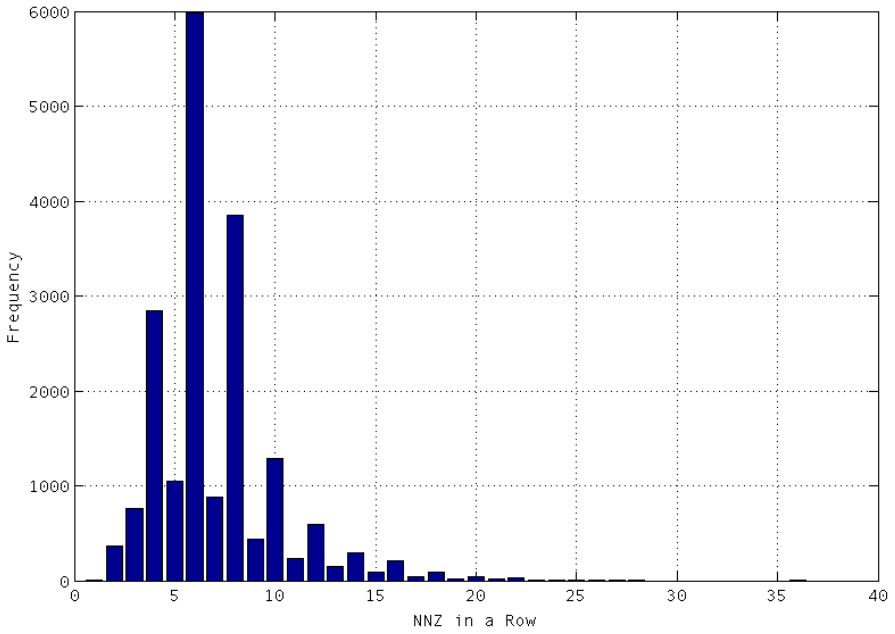

Figure 4.1: Distribution of NNZ per Row in the 10279 Bus Power System

have a small number of elements, but there are some larger submatrices.

Software measurements of the LU for power systems were obtained for the Intel Core i7 965 at 3.20GHz and an Intel Core2 Quad Q9300 at 2.50GHz. The execution times to perform the merges in each submatrix of the LU process were measured. The execution times, along with the number of rows in the submatrix and the length of each row, were output to a file. In addition, the total LU execution time was measured for each system with the Gaussian software and UMFPACK. These measurements are used to compare the performance of the LU software against the proposed architectures.

The average data rate on Core i7 at 3.2GHz, in millions of elements per second, for each power system is shown in Table 4.3. When compared to the clock frequency of the processor, it is clear that the merges are being performed very inefficiently for the power system matrices. The merge accelerator is capable of producing one output element per cycle, assuming there is enough input data available. Even at the lower clock rate needed for FPGA designs, the merge accelerator is capable of outperforming the software on power

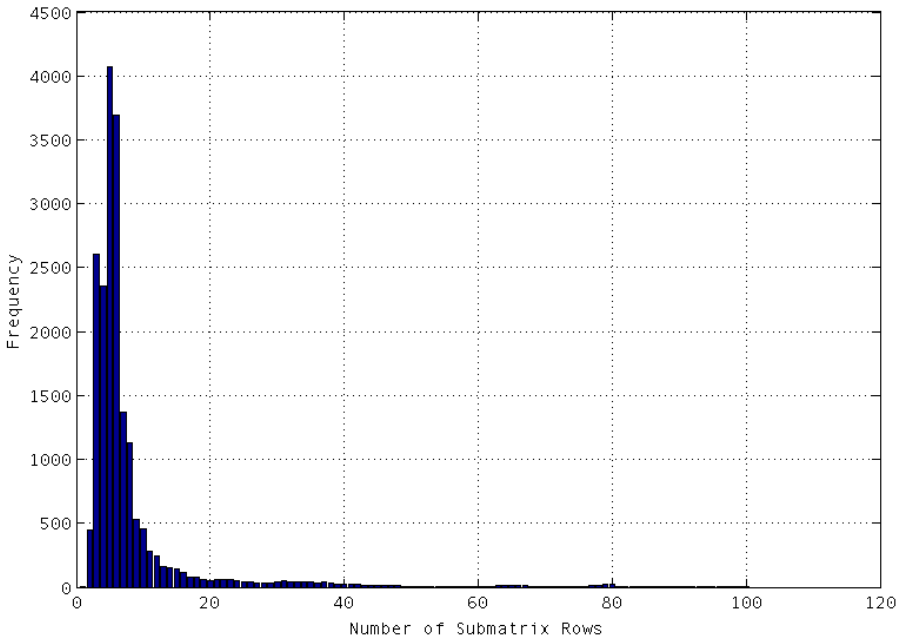

Figure 4.2: Distribution of Number of Rows per Submatrix in the 10279 Bus Power System

Table 4.3: Power Systems Average Merge Data Rate on an Intel Core i7 at 3.2GHz

Power System	Average Data Rate (Millions of Elements per Second)	Average Cycles per Element
1648 Bus	90.77	35.3
7917 Bus	99.54	32.2
10279 Bus	100.75	31.8
26829 Bus	97.46	32.8

Table 4.4: Random and Power System Matrix Properties

System	# Rows/Cols	NNZ	Sparsity
1648 Bus	2,982	21,196	0.238%
2K Random	2,982	20,288	0.228%
10278 Bus	19,285	134,621	0.036%
10K Random	19,285	127,211	0.034%

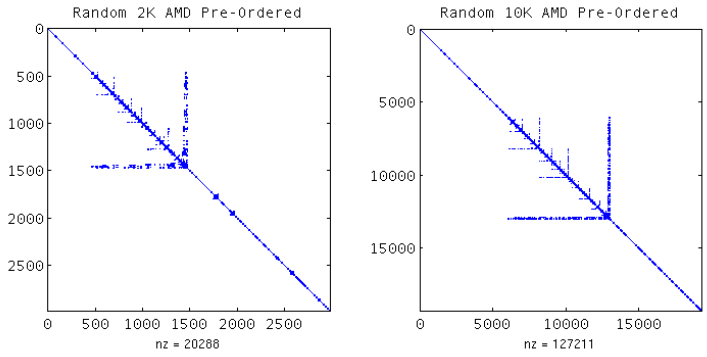

Figure 4.3: Non-Zero Structure of AMD-Ordered Random Matrices

systems. The challenge is being able to get the input data to the accelerator efficiently so that it provide speedup over the software merging.

4.2 Random Matrices

To demonstrate the unique properties of the power system matrices, random matrices with the same size and sparsity as the 2K-Bus and 10K-Bus power systems were generated. Table 4.4 shows the properties of the power systems and the randomly generated matrices of the same size. In addition, Figure 4.3 has the AMD-ordered non-zero structures of the random matrices. The random matrices do not have any elements along the edges of the matrix, as with the power systems, so it should be easier for UMFPACK to form frontal matrices.

Executing the random matrices with both UMFPACK and the Gaussian sparse LU re-

Table 4.5: Software Execution Times for Random and Power Matrices on Core i7 at 3.2GHz

System	UMFPACK Time (s)	# Frontal Matrices	Gaussian Time (s)
1648 Bus	0.007	1127	0.004
2K Random	0.004	673	0.005
10278 Bus	0.039	8032	0.030
10K Random	0.027	4336	0.313

Table 4.6: Properties of Select Matrices from UFSparse [7]

Name	# Rows/Cols	NNZ	Sparsity
aft01	8,205	125,567	0.19%
crystm01	4,875	105,339	0.44%
piston	2,025	100,015	2.44%

veals that the random matrices do not see the same benefit as the power matrices. Table 4.5 shows the execution times for the random matrices and power matrices on both UMFPACK and Gaussian LU. It also shows the number of frontal matrices formed by UMFPACK. With the random matrices, UMFPACK is able to outperform the Gaussian software on the Core i7 at 3.2GHz. UMFPACK is able to break the random matrices into fewer frontal matrices than the power systems, improving the performance of UMFPACK.

Figure 4.4 summarizes the speedup of Gaussian LU over UMFPACK for the power systems and random matrices. The power systems are able to outperform UMFPACK with Gaussian LU, but random matrices do not outperform UMFPACK with Gaussian LU. These results show that the actual structure of the power matrices, and not just the sparsity, contribute to the lower software performance on UMFPACK.

4.3 UFSparse Matrices

As a comparison to the power system matrices, three additional matrices were selected from the UFSparse collection [7]. Table 4.6 summarizes the properties of the three chosen matrices. The same LU software analysis was performed on the matrices. Table 4.7 shows

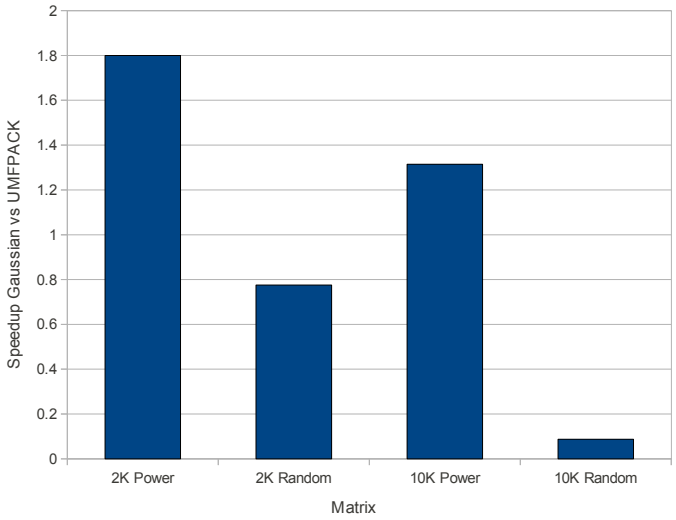

Figure 4.4: Speedup of Gaussian LU over UMFPACK for Random and Power Matrices on Core i7 at 3.2GHz

Table 4.7: UFSparse Matrix LU Analysis

Name	% Merge	Avg Rows per Submatrix	Avg NNZ per Row
aft01	77.2%	104.9	15.3
crystm01	72.0%	74.7	21.6
piston	67.7%	33.5	49.4

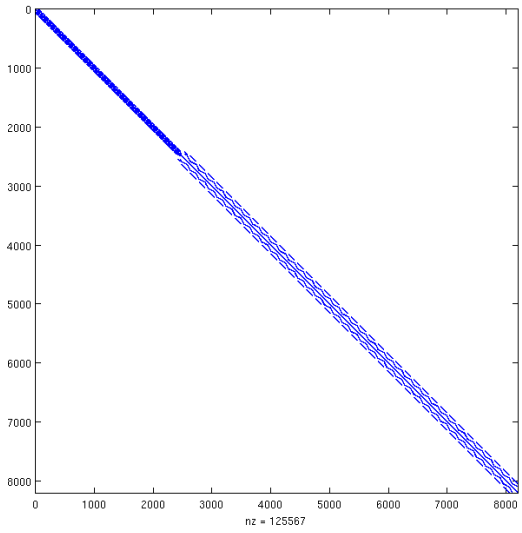

Figure 4.5: aft01 Non-Zero Structure

the percent of execution time spent doing the merging, the average number of rows per submatrix, and the average NNZ per row. These matrices are less sparse than the power systems, and have larger submatrices, on average. Figures 4.5, 4.6, and 4.7 show the non-zero structure of the aft01, crystm01, and piston matrices, respectively.

Unlike the power system matrices, these other sparse matrices do not perform better with the Gaussian software than UMFPACK. Because the matrices are not as sparse as the power matrices and have larger submatrices, the multi-frontal method of UMFPACK performs much better. UMFPACK is able to construct larger frontal matrices and take advantage of the high-performance dense matrix software routines. Figure 4.8 compares the performance of Gaussian LU and UMFPACK for the selected UFSparse matrices. UMFPACK performs much better than the Gaussian LU for these three matrices.

Table 4.8 shows the average merge data rates for these matrices on the Core i7 at 3.2GHz. There matrices perform better than the power matrices, in terms of merge performance. The processor spends fewer cycles on each element.

With these matrices, the merge accelerator can still provide a benefit over software, but it requires the accelerator to operate at a higher clock rate to achieve the same benefit as

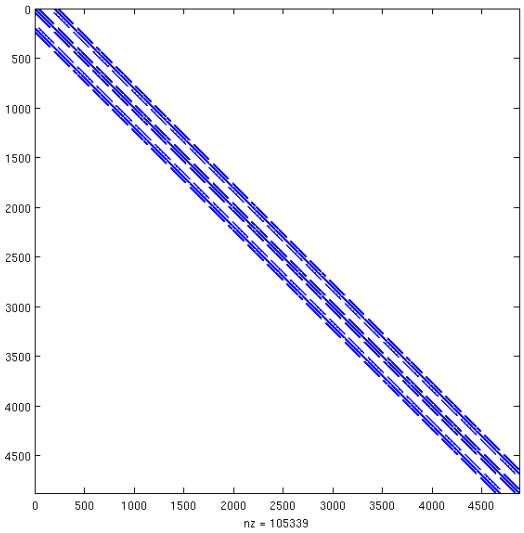

Figure 4.6: crystm01 Non-Zero Structure

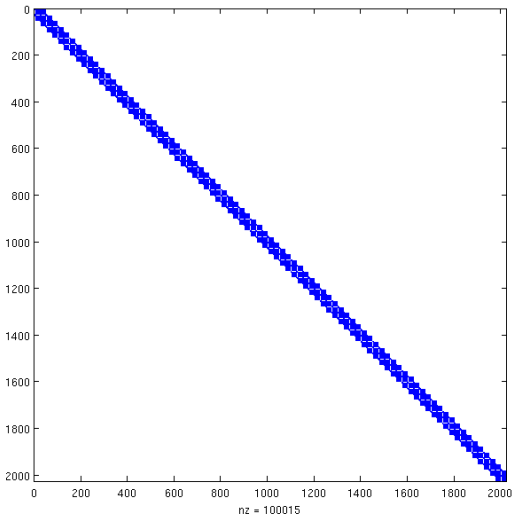

Figure 4.7: piston Non-Zero Structure

Table 4.8: UFSparse Average Merge Data Rate on an Intel Core i7 at 3.2GHz

Matrix	Average Data Rate (Millions of Elements per Second)	Cycles per Element
aft01	244.89	13.07
crystm01	232.68	13.75
piston	221.13	14.47

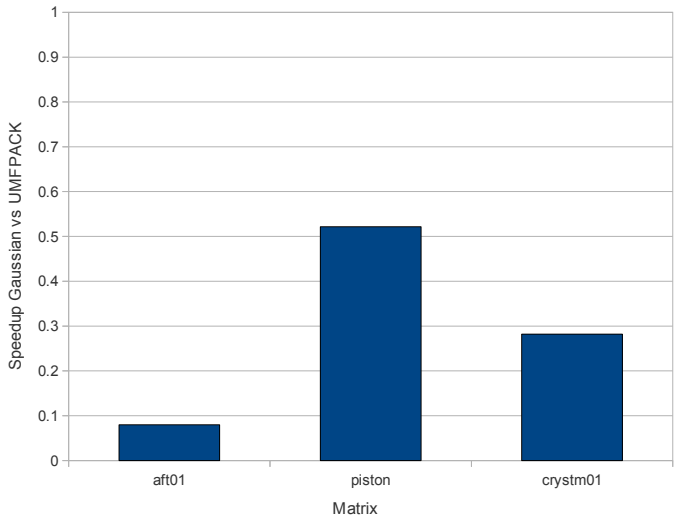

Figure 4.8: Speedup of Gaussian LU over UMFPACK on UFSparse Matrices [2]

with the power matrices. Because these matrices perform better in software, it will be more difficult to achieve better performance with the accelerator, because the data has less time to be sent to the accelerator, processed, and returned.

5. Triple Buffer Performance Analysis

The Triple Buffer Architecture attempts to fully utilize the transfer bus between a processor and FPGA by buffering blocks of data. Using the merge accelerator in the FPGA, the processor can send blocks of rows to the accelerator to be processed.

The Triple Buffer Architecture was implemented on the DRC board, which has a Xilinx Virtex 4 FPGA. The DRC hardware limits the clock frequency of the FPGA to 200MHz. The DRC board communicates with the processor across the HyperTransport (HT) bus. Since the DRC board does not have three banks of memory to serve as the buffers, the three buffers were implemented with BlockRAM FIFOs. Because these FIFOs operate at the FPGA clock rate and all data must pass in and out of a single connection to the HyperTransport bus, the bandwidth in and out of the buffers is limited. An actual implementation would provide a much better bandwidth between the processor and buffer, allowing the processor to upload and download data faster.

With the merge accelerator running at 200MHz, the prototype of the Triple Buffer Architecture was timed with blocks of data of varying sizes. Using the measurements of the bus bandwidth and processing times at different sizes, a performance simulation of the system was created. This simulation allows the performance of the system using other transfer buses to be predicted.

Section 5.1 analyzes the performance of the Triple Buffer Architecture for the power system matrices, while Section 5.2 analyzes other sparse matrices from the UFSparse collection.

5.1 Power System Matrices

Assuming the processor always has available data to send allows the maximum throughput of the system to be measured. Figure 5.1 shows the data rate in millions of elements per second (MEPS) for blocks with an increasing number of elements. For smaller block sizes, the overhead of transferring is significant when compared to the transfer time, caus-

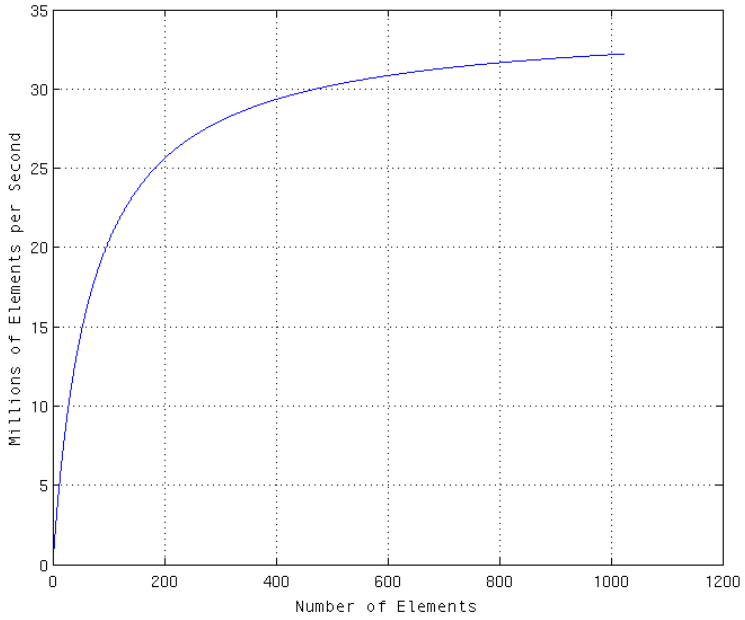

Figure 5.1: Merge Data Rate on DRC at 200MHz Using HT v1.0 Bus

ing low performance. As the block size increases, the transfer bus is better utilized and the data rate increases until it reaches a maximum performance of around 32 MEPS. This performance is very low compared to the power systems executing in software.

Considering the average block size of the power system matrices, the results suggest very poor performance for this architecture. Table 5.1 estimates the average data rate for each of the power systems based on the average number of elements in a submatrix. For these small block sizes, the power systems do not come close to the theoretical maximum for this system configuration, which is already well below the necessary performance.

These results also assume that there is always data to send to the accelerator. With sparse LU, the updated data from the previous iteration may be needed in the next iteration. In these cases, the accelerator must cache the updated row to use on the following iteration and the processor must know not to send the old data to the accelerator. While this is possible, it creates additional overhead and complexity in the design. For power systems, the Triple Buffer Architecture is not expected to perform well.

Table 5.1: Power System Data Rate

Power System	Average Elements Per Submatrix	Average Data Rate (MEPS)
1648 Bus	151	7.4
7917 Bus	208	9.4
10279 Bus	257	11.0
26829 Bus	382	14.1

Table 5.2: Power System Merge Speedup

Power System	Percent of Submatrices Using Accelerator	Merge Speedup
1648 Bus	0.000%	1.00000X
7917 Bus	0.007%	1.00001X
10279 Bus	0.005%	1.00002X
26829 Bus	0.002%	1.00000X

One improvement to this current method is to only send larger submatrices to the accelerator, while merging the smaller ones in software. This means that the software does not suffer from the poor performance of the accelerator on smaller blocks, but can benefit on larger blocks of data. Taking this approach, the performance increase on the merge section of the LU algorithm can be estimated by comparing the software merge time with the accelerator merge time. Table 5.2 shows the estimated merge speedup and percent of blocks that would use the accelerator for the power systems.

The merge speedup for the power systems is negligible. The accelerator is used so infrequently that it offers no performance benefit. Using the simulation model, the performance of the system with a faster transfer bus can be projected to see if there is any performance increase. The DRC board used for the prototype uses the HyperTransport v1.0. The newer HyperTransport v3.1 provides a 16X increase in bandwidth over the HT v1.0 configuration used in the DRC [4]. Figure 5.2 shows the estimated maximum performance of the Triple Buffer Architecture with the merge accelerator running at 200MHz and data always available. With the faster bus, the data rate approaches the clock frequency of the accelerator

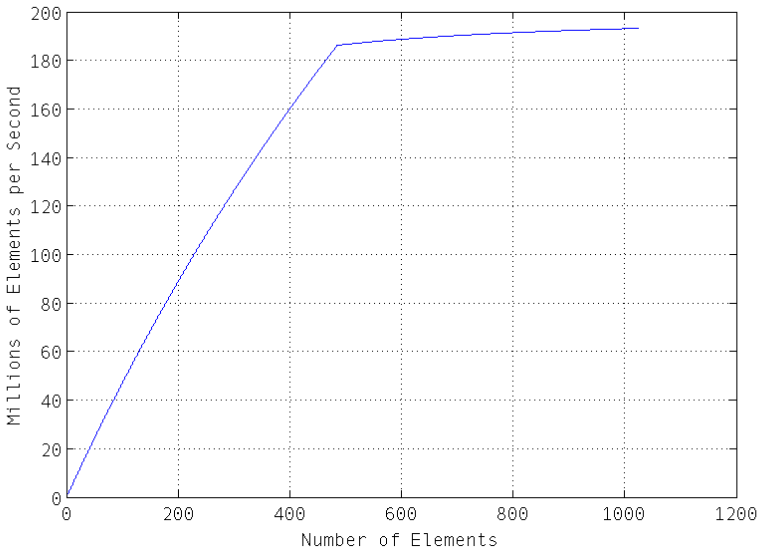

Figure 5.2: Merge Data Rate at 200MHz Using HT v3.1 Bus

Table 5.3: Power System Data Rate Using HT v3.1

Power System	Average Elements Per Submatrix	Average Data Rate (MEPS)
1648 Bus	151	9.3
7917 Bus	208	12.8
10279 Bus	257	15.7
26829 Bus	382	23.2

for larger blocks of data.

Even with the faster bus and much higher maximum performance, the average data rates for the power systems is much lower than necessary. Table 5.3 shows the average expected data rates for the power systems. The data rates using HT v3.1 are faster, but still far below the necessary performance for the power systems.

Because the performance is still too low for sending all submatrices to the accelerator, sending only the larger submatrices may be beneficial. Table 5.4 shows the expected merge speedup for the power systems. The HT v3.1 bus provides slightly better performance, but the accelerator utilization is still too low to provide any significant speedup.

Table 5.4: Power System Merge Speedup Using HT v3.1

Power System	Percent of Submatrices Using Accelerator	Merge Speedup
1648 Bus	1.51%	1.004X
7917 Bus	1.41%	1.003X
10279 Bus	0.78%	1.002X
26829 Bus	0.86%	1.003X

Table 5.5: Power System Data Rates with Zero Transfer Overhead

Power System	Average Elements Per Submatrix	Average Data Rate (HT v1.0) (MEPS)	Average Data Rate (HT v3.1) (MEPS)
1648 Bus	151	34.4	68.8
7917 Bus	208	34.4	83.9
10279 Bus	257	34.4	94.3
26829 Bus	382	34.4	114.0

For power systems, the Triple Buffer Architecture is not suitable for implementing the merge accelerator. The overhead of transferring the small blocks of data across an external bus limits the performance and eliminates any performance benefits from the accelerator.

As an additional experiment, the expected performance can be estimated if there is no transfer overhead. Table 5.5 shows the expected data rates for the HT v1.0 and HT v3.1 with no transfer overhead. The HT v1.0 transfer bandwidth limits the expected data rate. All of the power matrices reach this upper limit, which is well below the necessary performance. HT v3.1 provides a faster transfer and higher data rates. For the larger power systems, it is able to match or slightly exceed the performance of the software.

Even by considering no transfer overhead, the power Triple Buffer Architecture does not provide the necessary performance for sparse LU on power systems.

5.2 UFSparse Matrices

The small submatrix size of the power systems limits the data rate on the Triple Buffer Architecture. Other sparse matrices with larger submatrices may perform better than the

Table 5.6: UFSparse Data Rates

Matrix	Average Elements Per Submatrix	Average Data Rate (HT v1.0) (MEPS)	Average Data Rate (HT v3.1) (MEPS)
aft01	1605	25.6	89.1
crystm01	1614	25.6	89.6
piston	1655	25.8	91.6

Table 5.7: UFSparse Merge Speedup

Matrix	% Using Accel (HT v1.0)	Merge Speedup (HT v1.0)	% Using Accel (HT v3.1)	Merge Speedup (HT v3.1)
aft01	0.00%	1.000X	0.24%	1.000X
crystm01	0.00%	1.000X	0.47%	1.000X
piston	0.00%	1.000X	3.41%	1.003X

power matrices. Table 5.6 shows the average number of elements in the submatrix and the expected data for the HT v1.0 and v3.1 for the UFSparse matrices.

The UFSparse matrices have larger submatrix sizes than the power systems and have better expected data rates, but still do not exceed the average software data rate. Considering the merge speedup when sending only the larger submatrices does not provide any better results. Table 5.7 shows the percent of submatrices benefiting from the accelerator and the merge speedup for both the HT v1.0 and HT v3.1. For the HT v1.0, the accelerator is not used at all. Because the software performs better with the UFSparse matrices, it is more difficult for the accelerator to provide an increase over software. For HT v3.1, the accelerator is used slightly more, but still provides a negligible merge speedup.

In both the power systems and UFSparse matrices, there is no benefit to using the accelerator in the Triple Buffer Architecture. Even though the system can theoretically obtain a much higher data rate, the submatrix sizes are too small to take full advantage of the architecture. The Triple Buffer Architecture is better suited for applications sending larger blocks of data to an accelerator for more complex operations than a simple merge. The amount of time that the data spends in the accelerator does not justify the overhead of transferring the data. Also the data dependent nature of the sparse LU algorithm makes it

difficult to send a continuous stream of data. Stream-processing applications that process larger amounts of data would see a better benefit from this architecture.

6. DRAA Performance Analysis

The Microblaze processor and merge hardware implemented on the Virtex 6 FPGA serves as a prototype of the DRAA architecture. Using the prototype, performance measurements were collected for a wide range of merge scenarios. The number of processor execution cycles was measured for the time to execute a block of merges. The number of rows and lengths of the rows were varied. Using the collected measurements, a predictive performance model, based on the number of rows and total number of elements, was generated.

For each submatrix in the LU execution, the software execution cycles are compared to the DRAA execution cycles to perform all of the merges within the submatrix. Using the number of rows and total non-zero elements in the submatrix, the DRAA time is calculated from the predictive model. Assuming the Microblaze is replaced with a high-performance processor and the DRAA is implemented to run at the clock frequency of the processor, the software-only execution cycles can be compared directly to the DRAA execution cycles. By maintaining a consistent clock frequency ratio between the accelerator and processor, the performance of faster systems can be predicted from the cycle count of the implemented prototype. During LU execution, the software can decide whether to use the accelerator or software to execute a block of merges. Selecting the minimum execution time between software-only and software with an accelerator calculates the optimal speedup for the accelerator over software.

The merge speedup using the DRAA is projected for both the Intel Core i7 at 3.2GHz and Intel Q9300 at 2.5GHz. Section 6.1 analyzes the results for the power system matrices, and Section 6.2 analyzes the UFSparse matrices.

6.1 Power System Matrices

Comparing the merge performance of the software and accelerator shows how much the accelerator can increase the LU performance by improving the merge performance. Figure

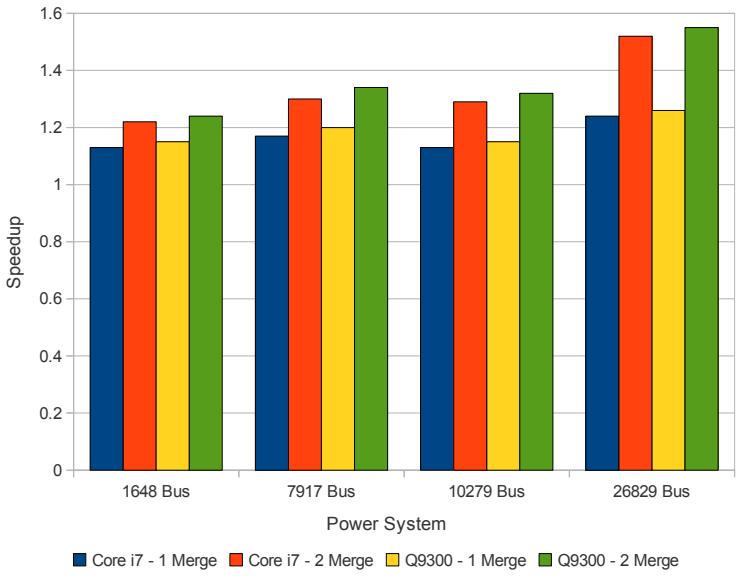

Figure 6.1: Merge Speedup vs Software-Only on Core i7 at 3.2GHz (Merge 320MHz) and Q9300 at 2.5GHz (Merge 250MHz)

6.1 displays the merge speedup for each power system on the two processors. The results are similar for both processors, but slightly higher for the Q9300. The Core i7 performs the merge process more efficiently than the Q9300, so it benefits slightly less from the accelerator. The Core i7 has larger caches and an integrated memory controller which likely provide the additional increase in performance.

As the system size increases, the number of submatrix rows and the number of non-zero elements in the submatrix increases, making the DRAA DMA transfers more efficient. Therefore, the larger systems see more of a benefit from the accelerator. With a single merge core, the reconfigurable hardware is only sending outputs on about half of the available cycles. By adding a second merge unit, the output port is almost fully utilized, allowing for a lower execution time and greater speedup.

Considering the percent of the execution time spent in the merge function, the total LU speedup from improving the merge execution time can be projected. Figure 6.2 shows the LU speedup for each processor. The LU speedups follow the same pattern as the merge speedups, where there is more benefit with larger systems and with two merge units. The

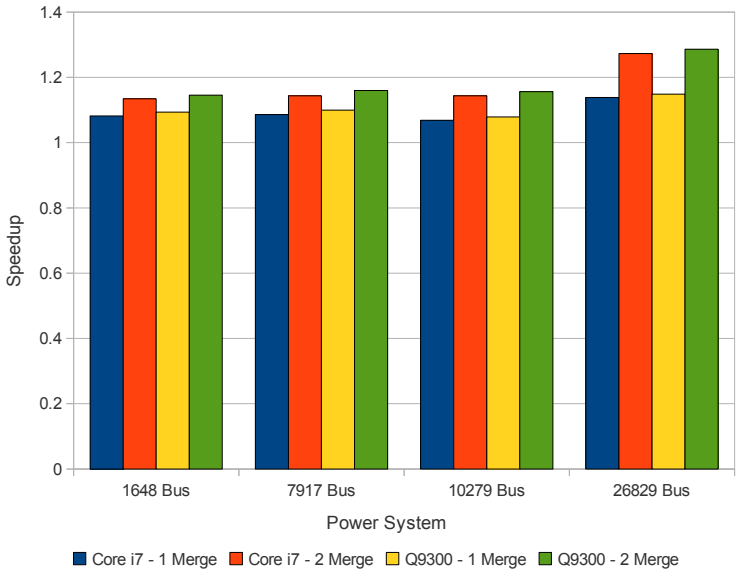

Figure 6.2: LU Speedup vs Software-Only on Core i7 at 3.20GHz (Merge 320MHz) and Q9300 at 2.5GHz (Merge 250MHz)

26K-bus system has the largest speedup, which is about 1.29X with two merge units.

While using the merge accelerator does provide a speedup over software-only execution, the performance is not close to the maximum speedup. A couple factors contribute to the reduced performance. First, the use of the Microblaze processor in the prototype provides less efficient software performance than a more advanced processor, such as the Core i7 or Q9300. These processors are able to execute instructions out-of-order and issue multiple instructions per cycle, while the Microblaze implements a simple, single-issue, in-order pipeline. The time to setup DMA transfers on the DRAA is a significant part of the total execution time. With a more efficient processor, these transfers could be setup quicker and improve the overall execution of the system. Therefore, the execution with the Microblaze processor provides a conservative view of the performance gain from using an accelerator.

The other limiting factor of the accelerator performance is caused by the power data itself. For the DMA architecture, the most benefit is when processing long rows. For short rows, the time to setup the transfer overshadows the transfer and computation time with the accelerator. With longer rows, the bus bandwidth is more efficiently utilized and the benefit

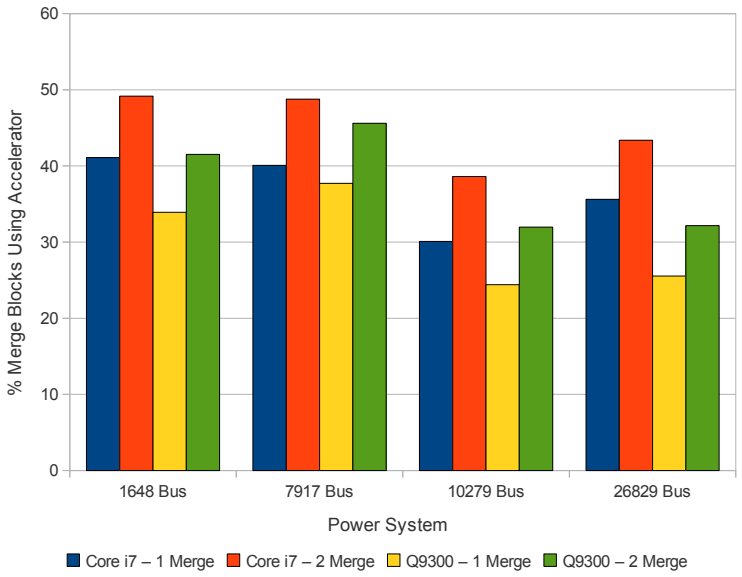

Figure 6.3: Percent of Submatrices Using Accelerator on Core i7 at 3.20GHz (Merge 320MHz) and Q9300 at 2.5GHz (Merge 250MHz)

from using the accelerator increases. As the power systems size increases, the potential for larger submatrices increases. However, the average number of rows in each submatrix only slightly increases with the larger systems. While there are some longer rows and larger submatrices in the larger systems, the majority of submatrices are fairly small. Figure 6.3 shows the percentage of submatrices where the accelerator outperforms the software on each system. As the system size increases, there are more submatrices to process, but the average size stays about the same, decreasing the percentage of submatrices that benefit from the accelerator. However, the large systems also have some larger submatrices which provide a very good speedup. This explains how the 26K-bus system achieves the highest speedup with the lowest percentage of submatrices being sent to the accelerator.

Running the reconfigurable hardware at 10 times slower than the processor gives a realistic balance between the processor and FPGA clock frequencies. As FPGA technology advances, the ratio between the processor and FPGA speeds can be further reduced. To explore the potential gain from having a relatively faster FPGA, the prototype was configured so that the FPGA runs only 5 times slower than the processor and the merge experiments

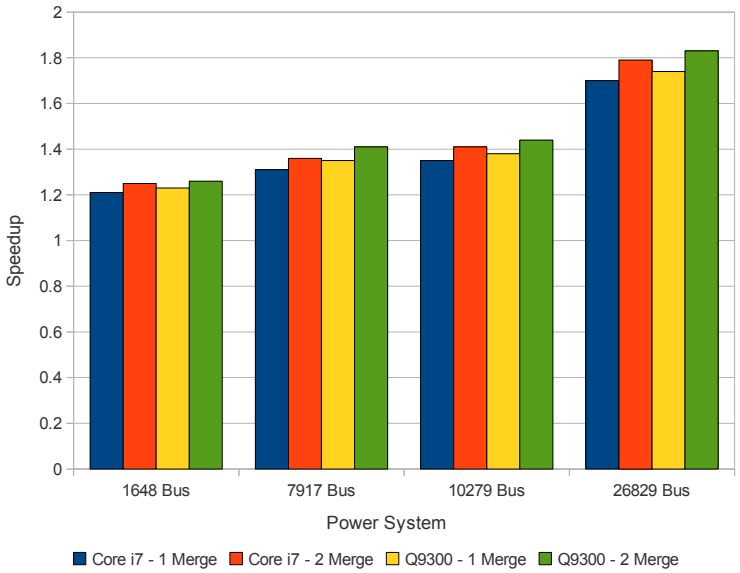

Figure 6.4: Merge Speedup vs Software-Only on Core i7 at 3.20GHz (Merge 640MHz) and Q9300 at 2.50GHz (Merge 500MHz)

were repeated. For the Core i7 at 3.2GHz, the accelerator now runs at 640MHz, and for the Q9300 at 2.5GHz, the reconfigurable hardware runs at 500MHz. Figure 6.4 shows the merge speedup, while Figure 6.5 shows the LU speedup for the accelerator running 5 times slower than the processor. Figure 6.6 shows the percentage of submatrices where there is a benefit to using the accelerator.

While doubling the accelerator frequency does provide a speedup, the performance does not double, because all of the data is still going through a single port to the accelerator. With the accelerator running faster, there is more strain on the system because it must supply the stream of data to the accelerator faster to keep it occupied. The overhead of setting up the DMA transfers and sending the data to the accelerator remains the same, so the performance benefit of the faster accelerator is reduced. The faster accelerator is able to provide as much as a 1.4X LU speedup on the 26K-bus system. There is a slight increase in the percentage of submatrices that benefit from using the accelerator. The accelerator still does not provide a benefit for the smaller submatrices, which are the majority in power systems.

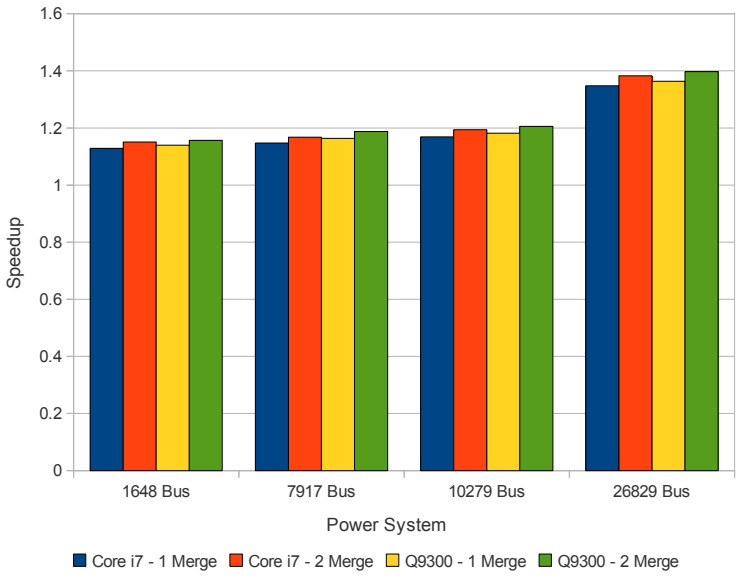

Figure 6.5: LU Speedup vs Software-Only on Core i7 at 3.20GHz (Merge 640MHz) and Q9300 at 2.50GHz (Merge 500MHz)

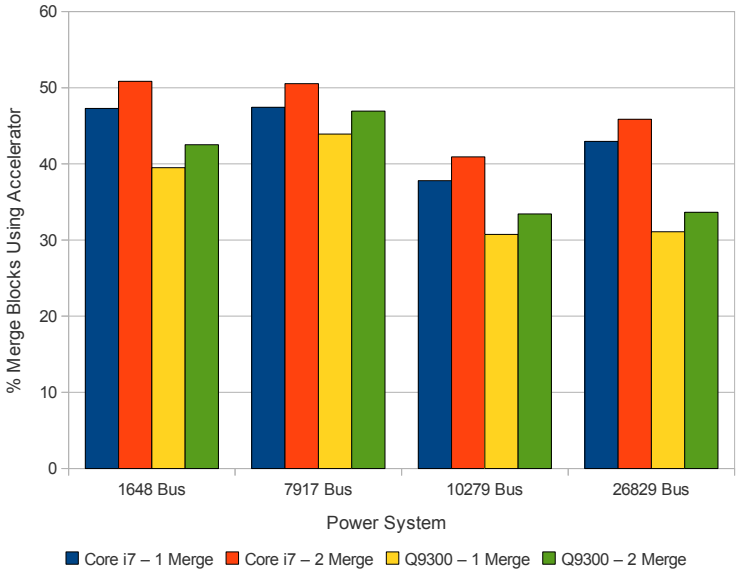

Figure 6.6: Percent of Submatrices Using Accelerator on Core i7 at 3.20GHz (Merge 640MHz) and Q9300 at 2.5GHz (Merge 500MHz)

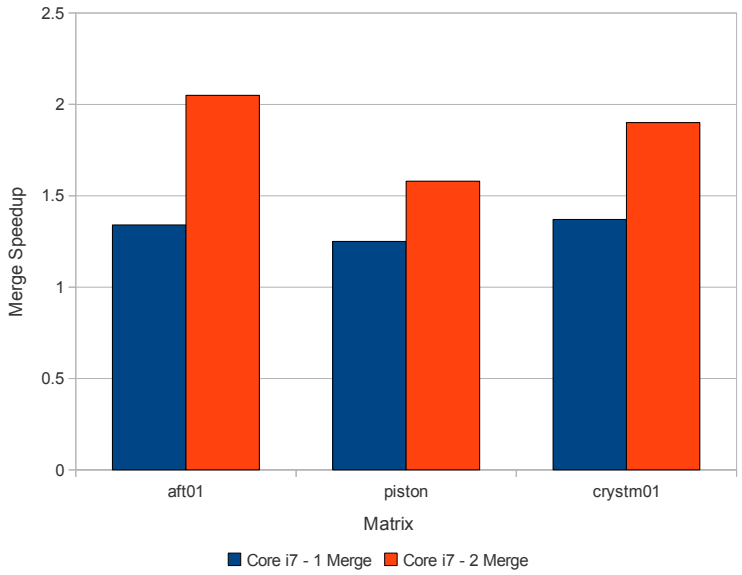

Figure 6.7: Merge Speedup vs Software-Only on Core i7 at 3.20GHz (Merge 320MHz) for UFSparse Matrices

6.2 UFSparse Matrices

Since the DRAA provides the most benefit for long rows and large submatrices, it may provide better performance on systems other than power systems. To explore the potential performance benefit on other systems, the three matrices from the UFSparse collection were also analyzed.

Using the same method as with the power systems, each of the matrices was analyzed for merge speedup, from using the merge accelerator. Because the matrices are less sparse and have larger submatrices than the power systems, the DMA architecture achieves a greater merge and LU speedup than the power systems. Figure 6.7 shows the merge speedup and Figure 6.8 shows the LU speedup over the Gaussian software on the Core i7 at 3.2GHz. Figure 6.9 shows the percentage of submatrices that benefit from using the merge accelerator. With these matrices, the DMA architecture is used on almost all of the submatrices, providing greater hardware utilization. In addition, the larger submatrices allows for more gain on each merge.

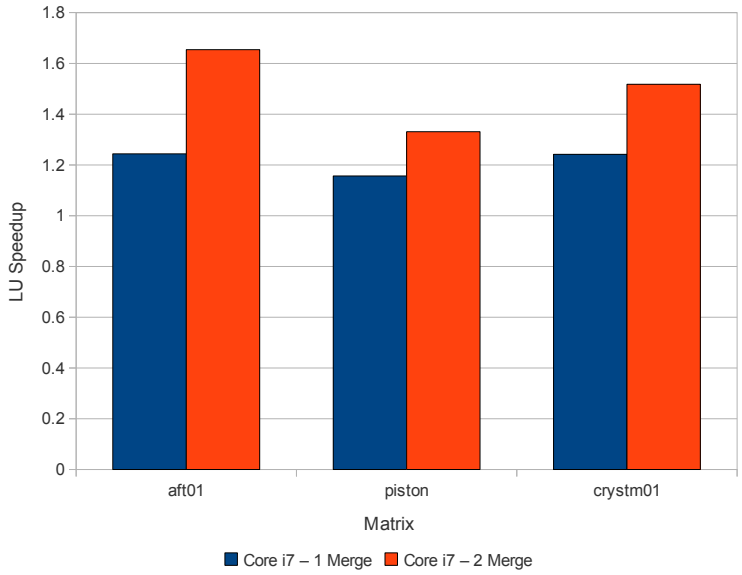

Figure 6.8: LU Speedup vs Software-Only on Core i7 at 3.20GHz (Merge 320MHz) for UFSparse Matrices

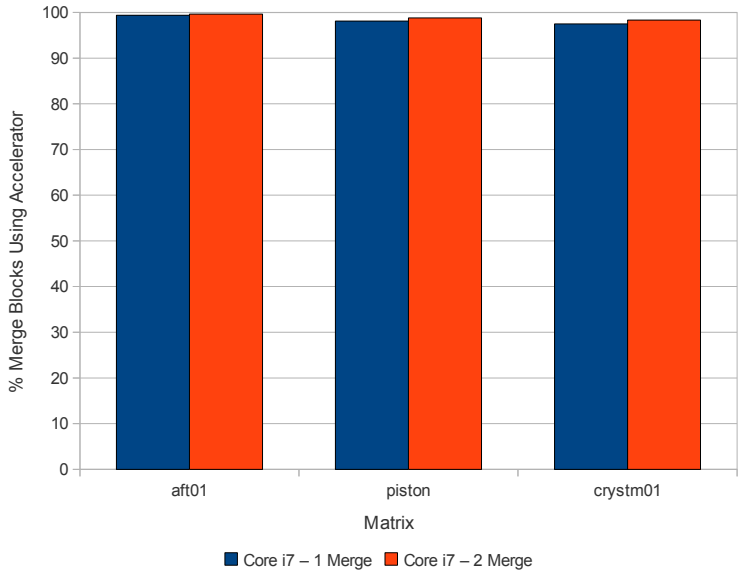

Figure 6.9: Percent of Submatrices Using Accelerator on Core i7 at 3.20GHz (Merge 320MHz) for UFSparse Matrices

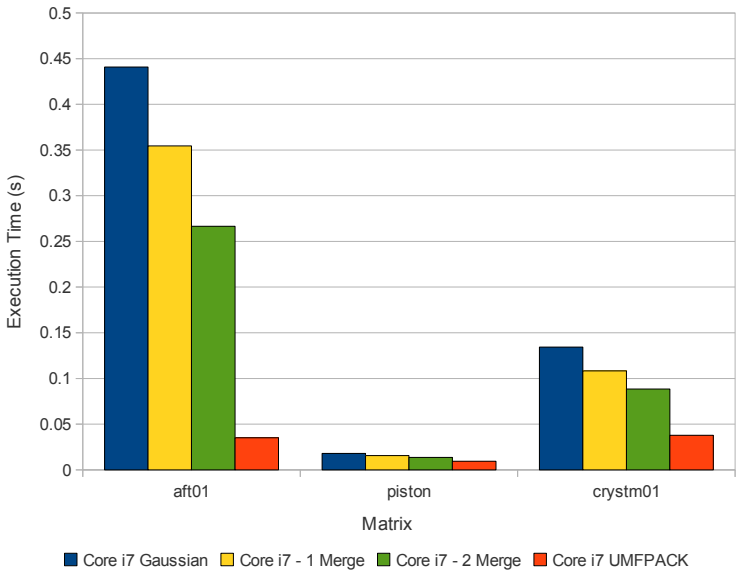

Figure 6.10: LU Execution Time on Core i7 at 3.20GHz for UFSparse Matrices

While these other matrices do perform better than the Gaussian software, the performance gain is less meaningful. With the power systems, the results rely on the fact that the Gaussian algorithm outperforms UMFPACK's multi-frontal method on power system matrices. However, the Gaussian algorithm does not see the same benefit with these other matrices. Figure 6.10 shows the execution times for the Gaussian method, the merge accelerator with one and two merge units, and UMFPACK on the Core i7 for each of the matrices. The DMA architecture provides significant gains over the software-only Gaussian method, but UMFPACK largely outperforms all of the Gaussian methods. The structure of these matrices helped both the accelerator and UMFPACK to perform more efficiently. The benefit of using the multi-frontal method is clearly seen with these matrices.

7. DPA Performance Analysis

The Gaussian sparse LU algorithm was implemented on the DPA architecture using the merge accelerator. The Data Processor (DP) loads the rows into local memory from external memory and stores the output rows back into external memory. The Compute Processor (CP) sends the rows to the merge hardware to process and updates the column map after rows are processed. The processors use a double buffering strategy to process the rows. The DP loads a block of rows to be processed into the first buffer and then loads another block of rows into the second buffer while the CP processes the first buffer. Similarly, the CP and DP alternate between two output buffers for the processed rows. In addition to merging a pair of rows, the merge unit also computes the pivot scale factor and fill-in columns. The scale factor is multiplied by each element in the pivot row, so that the leading elements of the two rows are eliminated. The scale factor is output into the L matrix. The fill-in columns determine which lists in the column map must be updated with the current row.

Section 7.1 explains the different configuration options for the DPA. Section 7.2 analyzes the performance of the power system matrices on the DPA and Section 7.3 analyzes the DPA performance with the UFSparse matrices.

7.1 DPA Configuration

The DPA allows for many different configuration parameters to be changed. The DP and CP frequency, DDR frequency, DDR bus width, DDR efficiency, and the overheads of different events in the system, are some of the available options. Table 7.1 lists some of the DPA parameters, along with the simulator defaults and the values chosen for the sparse LU experiments. While the vector unit was not used in the sparse LU design, the vector transfer delay was used to characterize the initial delay in transferring rows to and from the merge unit.

In addition to the DPA configuration options, the sparse LU design has some parameters

Table 7.1: DPA Architecture Parameters

Parameter	Default Value	Used Value
DP Frequency	1000 MHz	2000 MHz
CP Frequency	1000 MHz	2000 MHz
DDR Frequency	400 MHz	1066 MHz
DDR Bus Width	16 bytes	16 bytes
DDR Efficiency	1.0	1.0
CP Wait Overhead	4 cycles	4 cycles
CP Wait Loop Overhead	6 cycles	6 cycles
DP Wait Overhead	4 cycles	4 cycles
DP Wait Loop Cycles	6 cycles	6 cycles
CP Vector Transfer Delay	3 cycles	3 cycles

Table 7.2: DPA Merge Accelerator Frequencies

CP Frequency	Frequency Ratio	Merge Frequency
2000 MHz	10 times slower	200 MHz
2000 MHz	5 times slower	400 MHz
2000 MHz	2 times slower	1000 MHz
2000 MHz	Same frequency	2000 MHz

that can be adjusted. The block size determines the maximum number of rows that are loaded into an input or output buffer. For the following results, the default block size is 1, meaning that a single row is loaded into each buffer. The frequency of the merge accelerator can also be adjusted. The following results explore the performance with the merge unit operating 10 times slower than the CP, 5 times slower than the CP, 2 times slower than the CP, and at the same frequency as the CP. Table 7.2 provides the respective merge accelerator frequencies for a CP at 2000MHz. In the case of 5 and 10 times slower, the merge accelerator can be viewed as a reconfigurable hardware accelerator. However, to operate the merge accelerator at 1 or 2 GHz, it would need to be implemented as an ASIC. Therefore, the following results explore the benefit of using both a reconfigurable hardware and ASIC accelerator.

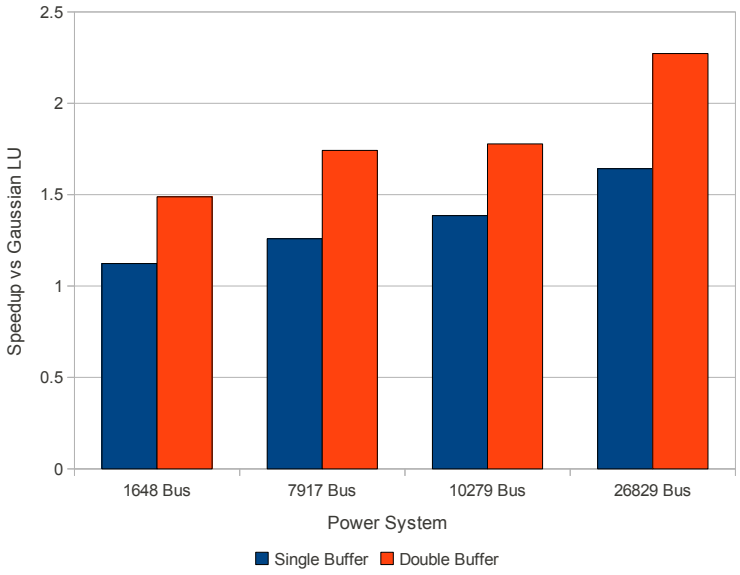

Figure 7.1: DPA Single vs Double Buffer Speedup Against Gaussian LU on Core i7 at 3.2GHz for Power Systems

7.2 Power System Matrices

The performance of the DPA sparse LU was compared against both the Gaussian and UMFPACK methods on an Intel Core i7 at 3.2GHz for the power systems.

One of the initial improvements over the basic sparse LU algorithm on the DPA is to use double buffering. For the DP, CP, and merge accelerator operating at 2GHz and the DDR frequency at 1066MHz, the speedup of the DPA over Gaussian LU on the Core i7 (3.2GHz) is shown for, both single and double buffering, in Figure 7.1. Double buffering provides an additional performance increase over the single buffering, because the DP and CP are able to process a second buffer instead of waiting for the other processor to finish processing.

Figure 7.2 shows the speedup against the Gaussian software for different merge accelerator frequencies, while Figure 7.3 shows the speedup against UMFPACK. These results use the DPA settings shown in Table 7.1, a block size of 1 row for the buffers, and double buffering.

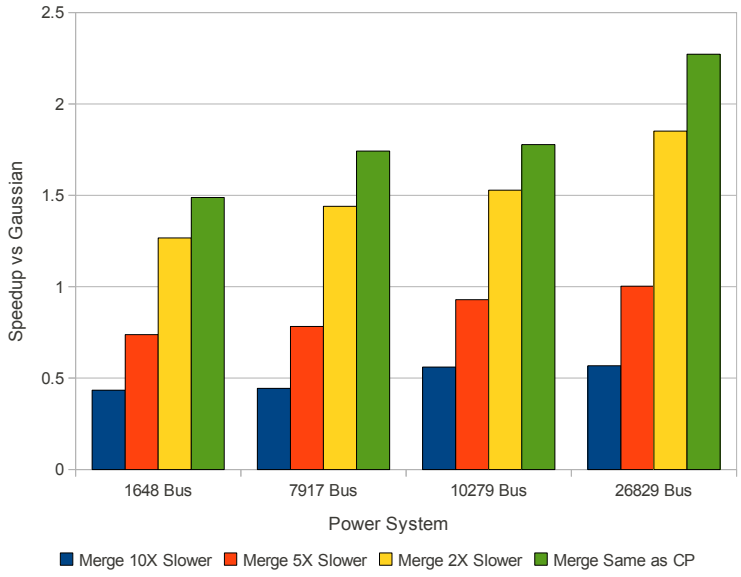

Figure 7.2: DPA Speedup vs Gaussian Software on Core i7 at 3.2GHz for Power Systems

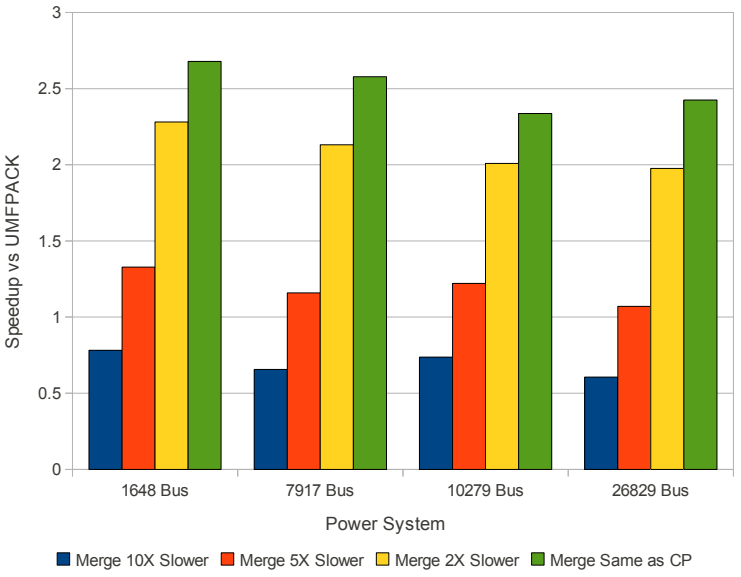

Figure 7.3: DPA Speedup vs UMFPACK on Core i7 at 3.2GHz for Power Systems

For the 10 times slower merge accelerator, both software versions outperform the DPA. The CP dominates the execution time because of the number of cycles spent in the merge accelerator. For the merge running 5 times slower, the DPA outperforms UMFPACK by 1.07X-1.33X, depending on the power system. However, the DPA does not perform better than the Gaussian software, except for a slight improvement in the 26K-Bus system. With the merge 2 times slower than the CP, the DPA provides a speedup over both software methods. The power systems have between a 1.98X and 2.28X speedup over UMFPACK, while the DPA provides between a 1.27X and 1.85X speedup over the Gaussian software. When the merge runs at the same frequency as the CP, the DPA sees between a 2.43X and 2.68X speedup over UMFPACK and 1.49X to 2.27X speedup over the Gaussian software.

With the faster merge, the DP execution time is becoming the bottleneck in the system, so further improvements to the merge speed provide a limited amount of additional speedup with the current configuration. Figure 7.4 confirms this barrier, by showing the CP execution cycles of the 26K-Bus system for each merge frequency. In addition, the CP execution time is shown for a zero execution time merge. The zero execution time merge provides little benefit over the merge unit running at the same frequency as the CP. Because the majority of the CP execution time is spent waiting for the DP to transfer to and from external memory, the merge time becomes insignificant in the overall execution time.

To further confirm that sparse LU is memory-bound and that the memory bandwidth is the bottleneck, the utilization of the memory bandwidth and the accelerator can be analyzed. Figure 7.5 shows the percent of cycles in which the DDR memory bandwidth and the accelerator are being used over the total execution of the 10K-Bus power system. The CP and DP are both running at 2GHz.

For the slower accelerator frequencies, the execution time is bound on the accelerator and the memory is used for a smaller percentage. As the accelerator frequency increases, the merge time decreases and the memory becomes the bottleneck in the system. For the faster accelerators, improving the memory bandwidth will allow for an increase in performance. Figure 7.6 shows the memory utilization as a percentage of the entire execution time for three DP frequencies on the 10K-Bus power system. The CP and accelerator are both

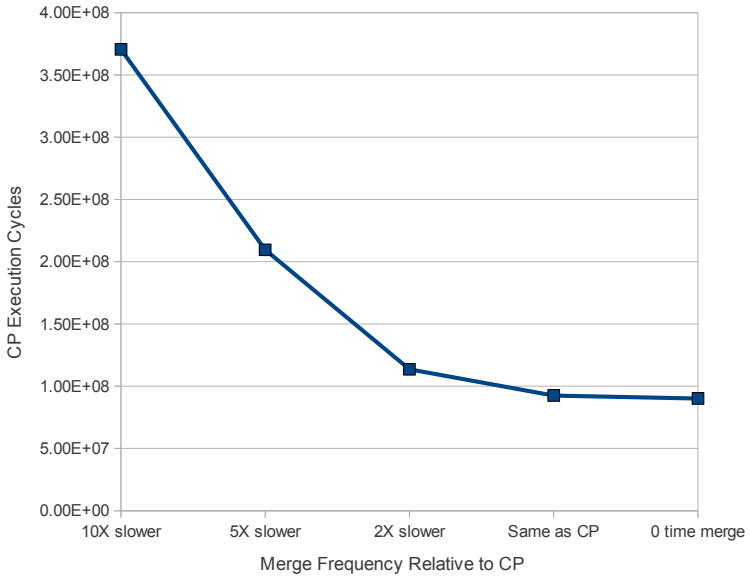

Figure 7.4: CP Execution Cycles for the 26K-Bus System at Different Merge Frequencies

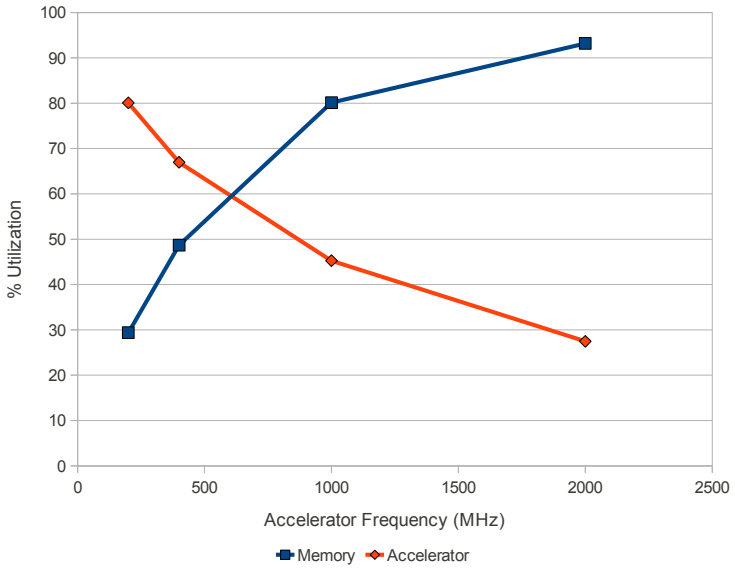

Figure 7.5: DPA Memory and Accelerator Utilization for 10K-Bus Power System

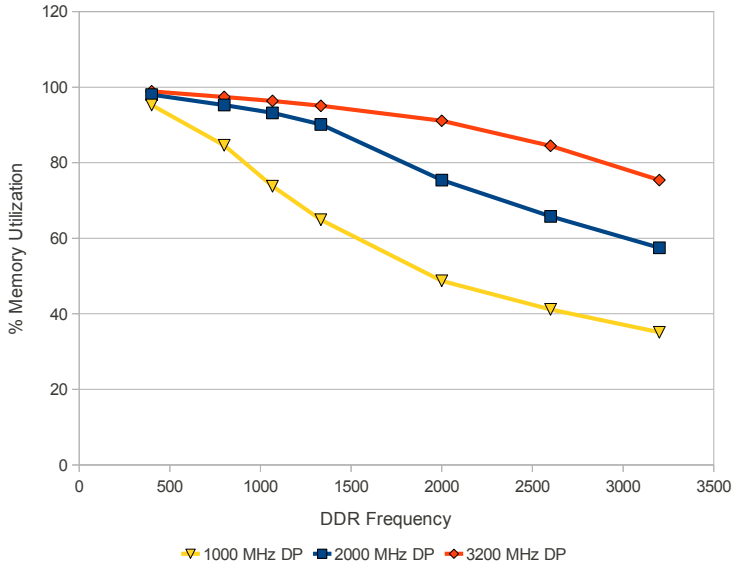

Figure 7.6: DPA Memory Utilization for Increasing DDR Frequency on 10K-Bus Power System

running at 2GHz in all cases.

As the DDR frequency increases, the memory utilization decreases. The slower frequency DP is not able issue memory requests fast enough to take full advantage of the faster DDR bandwidth, so its memory utilization decreases quicker than the higher DP frequencies. The DP at 3.2GHz is able to issue more requests as the DDR bandwidth increases maintaining a higher utilization and getting more data from the external memory. The result of the higher utilization can be seen in the execution times of the 10K-Bus system, shown in Figure 7.7. For the slower DDR frequencies the DPA is not able to outperform the LU software on the Core i7. As the DDR frequency increases, the total execution time decreases. The 1GHz DP sees less benefit than the higher frequency DPs, because of its lower memory utilization. As the DDR frequency continues to increase, there is less benefit in execution time. The execution time is therefore dependent on both the DDR frequency and the DP frequency.

The buffer block size for all of the above results is 1 row. Figure 7.8 shows the effect that changing the block size has on performance. For the 10K-Bus system, the LU performance on the DPA architecture was measured for block sizes up to 32 rows. Increasing the block

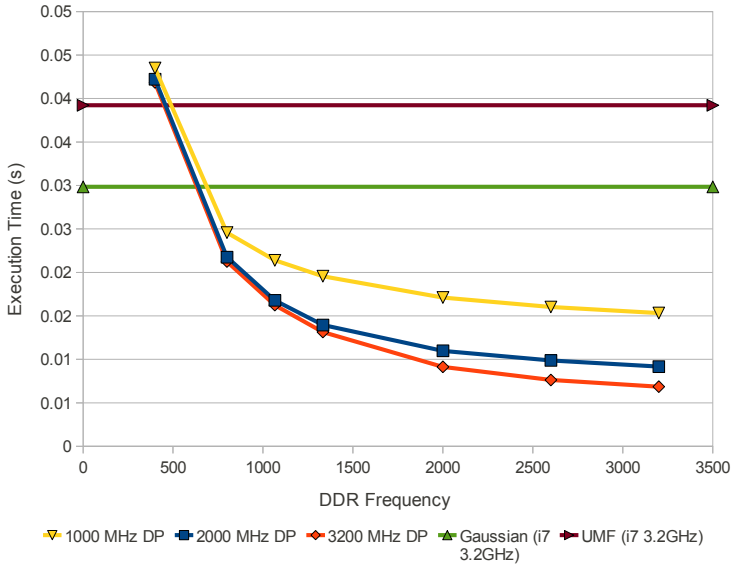

Figure 7.7: DPA Execution Time for Increasing DDR Frequency on 10K-Bus Power System

size above 1 increases the total execution time. The larger the buffer size, the longer the CP has to wait to begin processing the rows. Increasing the block size decreases performance, so it is best to use a single row in each buffer. This method also has the most scalability, since multiple long rows could potentially overflow the buffer.

Another method of improving the DPA performance is to increase the frequency of the processors. Increasing the processor frequency from 2GHz to 3.2GHz, the frequency of the Core i7, allows the processors to issue instructions faster and move data through the system faster. The merge accelerator frequency also increases, because it is based on the frequency of the CP. The frequency of the DDR remains at 1066MHz, which restricts how fast data can be accessed in external memory.

Figure 7.9 shows the DPA speedup over the Gaussian software, while Figure 7.10 shows the DPA speedup over UMFPACK with the DPA running the DP and CP at 3.2GHz. The DPA with the 10 times slower merge accelerator does not outperform the Gaussian software, but slightly outperforms UMFPACK for the 2K-Bus and 10K-Bus systems. The 5 times slower merge accelerator now outperforms both UMFPACK and Gaussian LU. For the 2 times slower and same as the CP merge accelerator, the external memory bandwidth

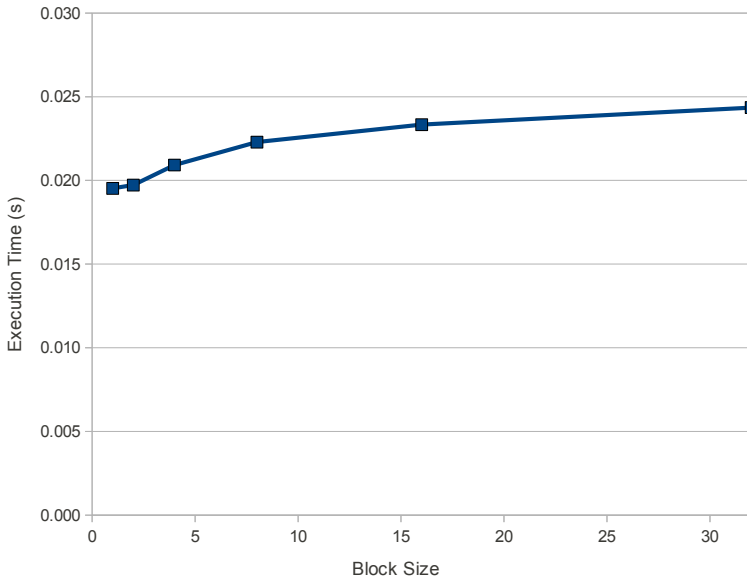

Figure 7.8: DPA Execution Time for 10K-Bus Power System with Varying Block Sizes

becomes the bottleneck, causing the speedup to be similar for both. Even though the DP is running faster, the DDR bandwidth limits how quickly the DP can move data from external memory to local memory. Therefore there is not as much improvement from increasing the accelerator frequency to the same as the CP from 2 times slower than the CP.

It is important to note that the 5 times slower merge accelerator for a 3.2GHz CP is running at 640MHz, which is outside the range of FPGA frequencies with current technologies, considering a Virtex 6 has a maximum frequency of 600MHz [24]. Even though the 5 times slower accelerator is providing greater performance than software, it now requires an ASIC merge accelerator.

7.3 UFSparse Matrices

The DPA provides a good speedup for the power system matrices with the faster running merge accelerator. This section examines the performance of the UFSparse matrices on the DPA with the merge accelerator.

Figure 7.11 shows the LU speedup on the DPA architecture against the Gaussian soft-

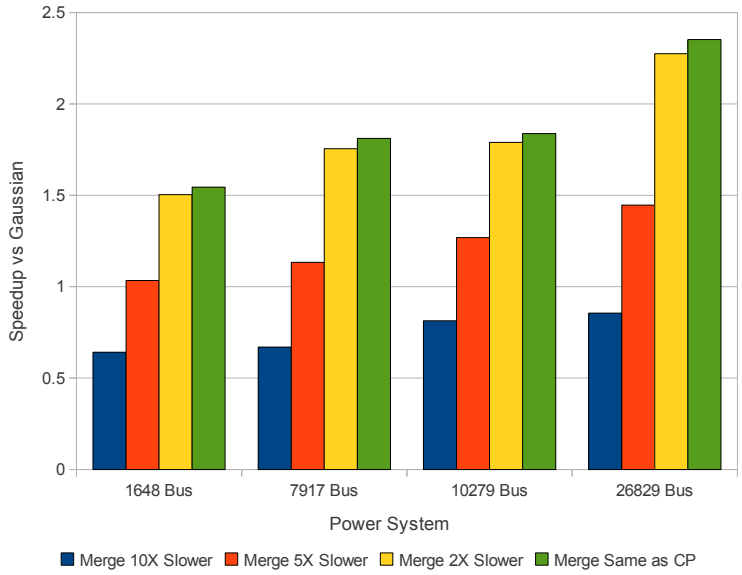

Figure 7.9: 3.2GHz DPA Speedup vs Gaussian Software on Core i7 at 3.2GHz for Power Systems

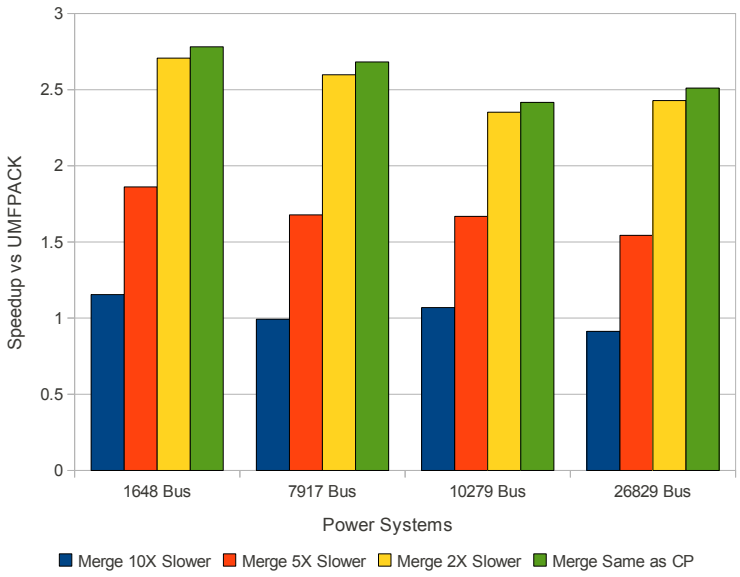

Figure 7.10: 3.2GHz DPA Speedup vs UMFPACK on Core i7 at 3.2GHz for Power Systems

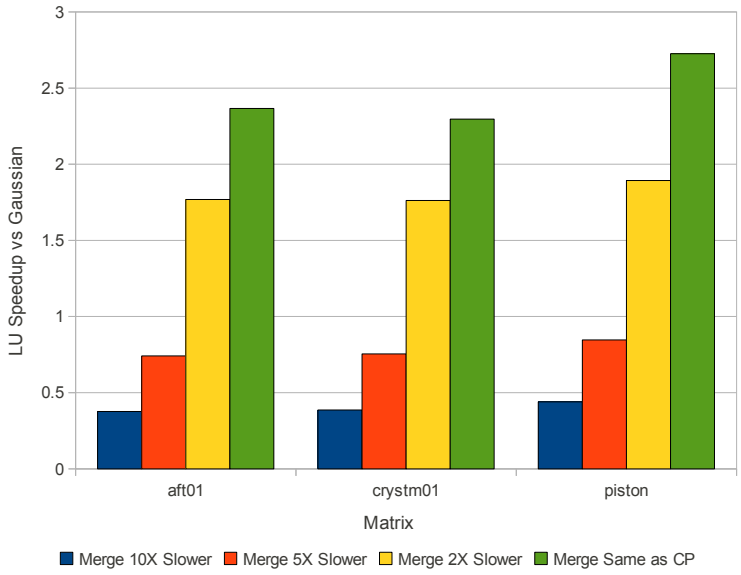

Figure 7.11: DPA Speedup vs Gaussian Software on Core i7 at 3.2GHz for UFSparse Matrices

ware for the UFSparse matrices. As with the power matrices, the slower merge accelerator does not improve performance, but the faster accelerators provide significant speedups. The piston matrix receives the most benefit from the DPA with the 2 times slower accelerator gaining a 1.9X speedup and the accelerator running the same frequency as the CP gaining a 2.7X speedup over Gaussian LU.

The DPA does not provide the same increase in performance against UMFPACK for the UFSparse matrices. Because the multi-frontal software performs much better with the UFSparse matrices, it is much more difficult for the DPA to outperform the software. Figure 7.12 shows the DPA LU speedup for the UFSparse matrices against UMFPACK. The aft01 and crystm01 matrices perform worse than UMFPACK on the DPA, even with the faster running accelerator. The piston matrix is the only one to see any benefit from the DPA. It is able to almost match the performance of UMFPACK with the 2 times slower accelerator and provide a 1.44X speedup with the accelerator running at the same frequency as the CP. In general, it does not seem beneficial to use sparse LU on the DPA for matrices other than power system matrices, or other similarly structured sparse matrices.

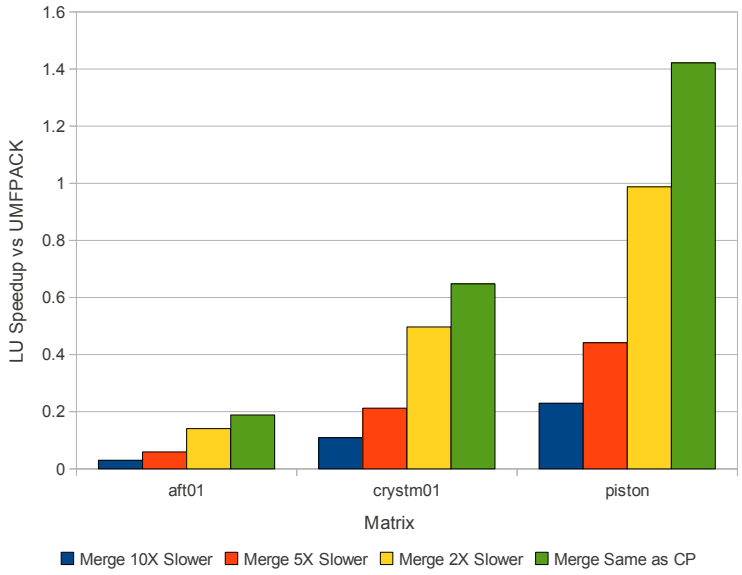

Figure 7.12: DPA Speedup vs UMFPACK on Core i7 at 3.2GHz for UFSparse Matrices

As with the power systems, increasing the frequencies of the CP and DP to 3.2GHz provides some additional performance. Figure 7.13 shows the speedup over Gaussian LU with the DP and CP running at 3.2GHz, while Figure 7.14 shows the speedup over UMFPACK. The UFSparse matrices still perform better with UMFPACK than on the DPA. Piston is the only matrix that exceeds UMFPACK on the DPA, but now outperforms UMFPACK with the 2 times slower merge accelerator as well.

By improving the execution time of the DP, the DPA could achieve even greater speedups. One method is to improve the DP and DDR configuration parameters to improve memory access performance. Another method is to implement a caching or reuse detection for rows in local memory. Sparse LU on power systems reuses many of the same rows over several iterations [18]. Taking advantage of this locality could reduce the number of memory accesses that the DP must make to external memory. Instead, output rows that will be used on the next iteration can stay in the local memory and be sent back to the merge by the CP.

A further improvement of the merge process, which could help the slower frequency

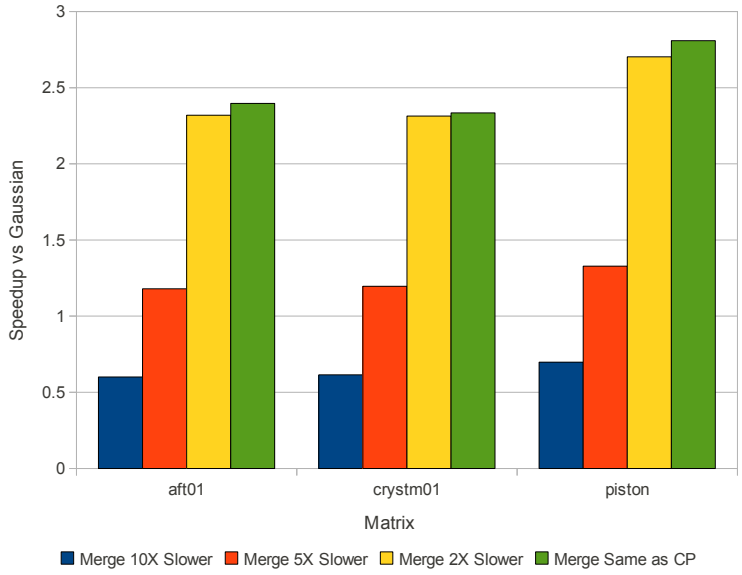

Figure 7.13: 3.2GHz DPA Speedup vs Gaussian Software on Core i7 at 3.2GHz for UFSparse Matrices

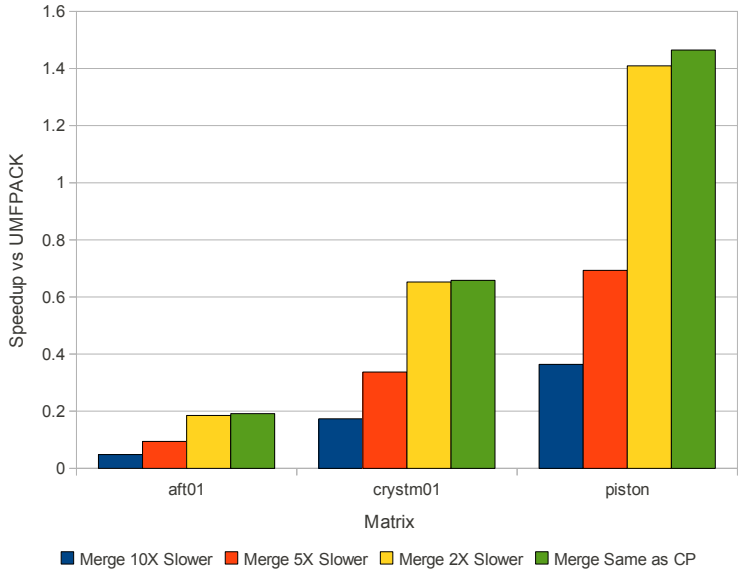

Figure 7.14: 3.2GHz DPA Speedup vs UMFPACK on Core i7 at 3.2GHz for UFSparse Matrices

merge accelerators to perform better, would process the merge on a block of rows. Currently the DPA assumes a single merge unit and performs each merge one after the other. Instead, the CP could take all of the rows in the buffer and send them to the merge accelerator. With multiple merge units, the accelerator can merge multiple pairs of rows in parallel, improving the CP execution time.

8. Performance Comparison

The previous chapters presented the performance results of sparse LU using a merge accelerator on three different architectures. The first architecture, the Triple Buffer Architecture, is based on the existing paradigm for many commercially available FPGAs boards. It uses a buffering strategy in an attempt to efficiently utilize the transfer bus between a processor and FPGA for streaming computation. The second architecture, the DMA Reconfigurable Accelerator Architecture (DRAA), combines a general-purpose processor and reconfigurable accelerator into a single package, allowing the two devices to share the same memory system. The processor controls the data going to the hardware accelerator by issuing DMA transfer requests to a DMA module that provides streams of data to the accelerator. The third architecture, the Data Pump Architecture (DPA), uses multiple processors to control the movement of data from memory to a hardware accelerator. The Data Processor (DP) moves data between external memory and local memory, while the Compute Processor (CP) moves data from local memory to the merge accelerator.

The Triple Buffer Architecture did not provide good performance for the application of sparse LU with a merge accelerator. The small blocks of data and simple merge unit make it inefficient to send data to an external FPGA. The short, data-dependent iterations of the LU algorithm make it more difficult to send continuous streams of data to the accelerator. This architecture is not suitable for sparse LU, but could provide strong performance to other applications that process larger batches of data and can make use of a more complex accelerator.

The DRAA and DPA both provided a performance increase over software for the power matrices, with the DPA showing the best speedup results. Figure 8.1 compares the execution times of the power systems on software, the DRAA, and the DPA. The software includes UMFPACK and Gaussian LU running on the Core i7 at 3.2GHz. The DRAA result is with a Core i7 processor at 3.2GHz and a 10 times slower merge accelerator with two merge cores. The DPA has both the DP and CP running at 2GHz with the merge accelerator

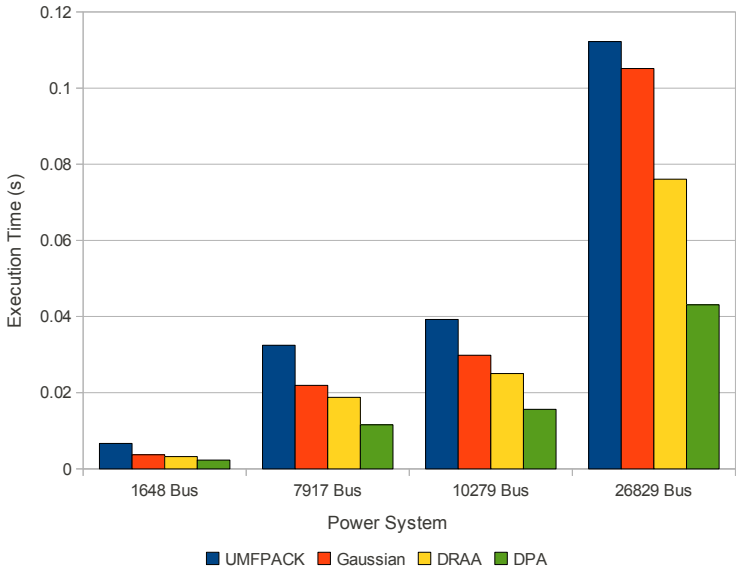

Figure 8.1: Power System Performance Comparison

running at the same frequency as the CP. For the power systems, UMFPACK has the longest execution time, followed by the Gaussian software. The DPA provides the largest decrease in execution time.

In general, the DPA provides more of a performance increase over software than the DRAA, but there are some important distinctions between the DRAA and DPA that provide additional context to the results.

The DRAA was able to achieve speedups over software with an accelerator running 10 times slower than the processor, while the DPA needed to run the accelerator at 2 times slower or the same frequency as the CP to see significant speedups. To achieve these frequencies, the DPA would need to implement a special purpose merge ASIC, losing the flexibility of having a reconfigurable accelerator that can be used for other applications. The DPA requires such a fast accelerator because it uses the merge accelerator for every submatrix, while the DRAA only sends submatrices that will benefit from using the accelerator. The DRAA executes most of the LU algorithm in software and only uses the merge accelerator for the merge operation when it was beneficial. The DPA had to send every submatrix, no

matter what size, through the merge accelerator.

Because the DRAA is only optimizing one section of the LU algorithm, the maximum potential speedup is limited by the amount of time spent doing the merge during LU. In comparison, the DPA ports the entire LU algorithm onto the DPA architecture. While porting sparse LU to the DPA requires more initial work, the DPA is able to potentially achieve greater speedups than the DRAA. The DPA can optimize the memory transfers and, since it does not have a cache, will not suffer from cache misses like the software. The fact that the DPA was able to achieve greater speedups while sending every merge through the accelerator, shows the level of optimization that the platform provides.

The DPA is a specially-designed architecture, while the DRAA is an extension to a general-purpose multicore architecture. A platform similar to the DRAA is more likely to be commercially available, while an implementation of the DPA would likely need to be custom-made. In the future, it is possible that architectures like the DPA could become commercially available, but a custom-designed ASIC for the merge accelerator would still need to be designed, fabricated, and integrated into the DPA in order to implement sparse LU. With Intel planning to release a product combining an FPGA and processor in a single package [11], it seems likely that general-purpose heterogeneous architectures will become popular in the coming years. While the DRAA performance gain is not as significant as the DPA performance gain, sparse LU seems more easily implementable, in the near future, using a general multicore platform.

In all of the architectures, the bottleneck is the bandwidth and overhead of transferring data to the accelerator. By improving the memory bandwidth, the performance of all of the architectures can be improved. In addition, implementing caching and prefetching of rows in a local memory will reduce the number of transfers and improve the performance.

The DPA and DRAA provided good results for power matrices, but the other sparse matrices did not perform as well. While the UFSparse matrices showed additional speedup over the Gaussian software on these architectures, UMFPACK overshadowed all of the results with very strong performance, as shown in Figure 8.2. The figure compares the software execution time of UMFPACK and Gaussian LU on the Core i7 at 3.2GHz with the DRAA

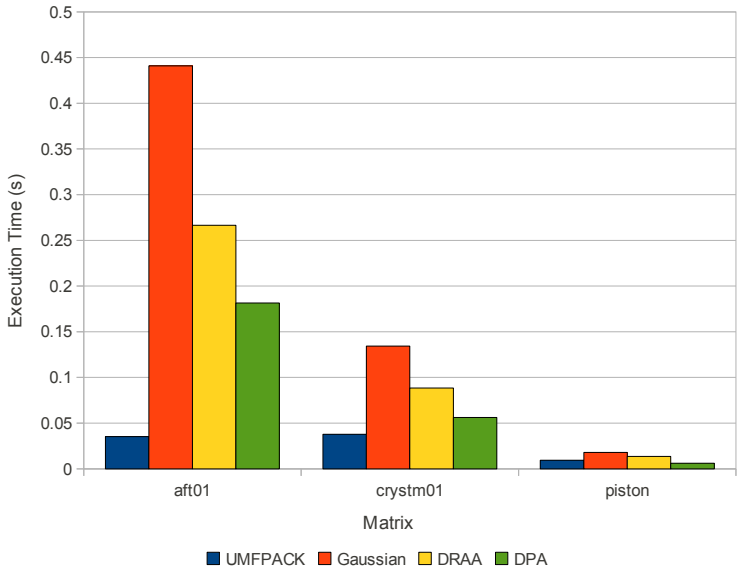

Figure 8.2: UFSparse Performance Comparison

and DPA execution times. The DRAA is assuming a Core i7 at 3.2GHz processor and a 10 times slower merge accelerator. The DPA is running the DP and CP at 2GHz and the merge accelerator at the same frequency as the CP. The only matrix to outperform UMFPACK is the piston matrix on the DPA. The DRAA and DPA both achieve a performance increase over the Gaussian LU software for the UFSparse matrices.

9. Conclusion

Sparse linear algebra, and specifically sparse Lower-Upper Triangular Decomposition, is important to many applications, including power system analysis. Utilizing custom hardware accelerators is one method of improving the computational performance of sparse applications. In order to effectively use accelerators, an efficient architecture that allows for low-overhead communication between a processor and accelerator is crucial.

This work demonstrates, with the case of the Triple Buffer Architecture, that some of the existing methods for connecting processors and FPGAs are not suitable for the close-coupled interaction necessary for sparse LU. Instead, architectures that combine the processor and reconfigurable hardware in a much more integrated way are necessary.

The DRAA explores the emerging heterogeneous multicore architectures by connecting a processor and the accelerator to the same memory system. By offloading the responsibility of fetching data from memory and providing it to the accelerator, the processor is free to perform other computations while the accelerator processes the data. The merge accelerator is able to speed up the merge portion of the sparse LU algorithm and provide a significant speedup over sparse LU software. The results on this architecture are relevant to the many other similar architectures combining processors with accelerators. The merge accelerator can be mapped onto other architectures and should provide similar results.

The DPA explores a different type of architecture where the processors explicitly manage the memory transfers and there is no automatic data caching as in general multicore architectures. While this platform can be more restrictive and require more effort to implement algorithms, it provides the opportunity for greater speedups than in a general multicore architecture. By having fine control over the data in local memory, the processors can keep the execution units efficiently utilized and does not have to suffer from data cache misses. The DPA also provides a good speedup for sparse LU on power systems even with lower processor clock frequencies than the general-purpose processors.

The downside is that none of the three architectures provided any significant results

for matrices other than power systems. Only a small number of other sparse matrices were tested, so there may be other sparse matrices that are structured similarly to power systems that will achieve the same speedup. However, it seems that software performs very well for a majority of sparse matrices.

Even though the results of these architectures cannot be generalized to all sparse LU problems, the architectures themselves are fairly general and can be applied to other applications. Replacing the merge accelerator with a different accelerator could provide similar improvements in other applications. The information about using sufficient blocks of data to process and streaming information to accelerators is applicable to many applications and architectures.

The LU speedups achieved by these architectures can provide significant time savings for the power flow calculations in contingency analysis. Because the power flow calculation is repeated thousands of times, even small savings will accumulate over time. Although these architectures do not provide large, order-of-magnitude speedups typically expected from accelerators, the increase in performance can still be beneficial for such a small development effort. Since general multicore architectures will likely provide reconfigurable cores in the future, sparse LU could easily be implemented on these platforms with little effort. The increase in performance can allow for faster grid analysis, so that potential faults can be handled sooner.

9.1 Future Work

While the DRAA and DPA showed good results for sparse LU, there is potential for even more improvement. None of the presented implementations used data caching for the accelerator. By detecting rows that will reused on the next iteration and keeping these rows in a local memory, the number of memory transfers can be reduced. Investigation of reuse caching could improve the performance of the presented architectures. In addition, testing other configurations for component frequencies in both the DRAA and DPA could help to determine more realistic estimates.

It could also be useful to explore using the merge accelerator on other architectures

combining processors and accelerators. Other architectures may be able to provide better performance than the already presented results. In addition, implementing sparse LU on an actual hardware device will provide more concrete performance results.

Lastly, using these architectures to explore other applications could prove useful. Developing a library of different accelerators that could be used on a single accelerator platform will take advantage of the reconfigurable hardware aspect and allow the same architecture to be used for many different applications.

Bibliography

[1] A. Albur, A., Expsito. *Power System State Estimation: Theory and Implementation.* Marcel Dekker, New York, New York, USA, 2004.

[2] T. Chagnon. *Architectural Support for Direct Sparse LU Algorithms.* Masters thesis, Drexel University, 2010.

[3] DRC Computer. DRC Coprocessor System User's Guide, 2007.

[4] HyperTransport Consortium. HyperTransport I/O Link Specification: Revision 3.10C, 2010.

[5] Kevin Cunningham and Prawat Nagvajara. Reconfigurable Stream-Processing Architecture for Sparse Linear Solvers. *Reconfigurable Computing: Architectures, Tools and Applications*, 2011.

[6] Kevin Cunningham, Prawat Nagvajara, and Jeremy Johnson. Reconfigurable Multicore Architecture for Power Flow Calculation. In *Review for North American Power Symposium 2011*, 2011.

[7] Timothy Davis. University of Florida Sparse Matrix Collection.

[8] Timothy Davis. Algorithm 832:UMFPACK V4.3 - An Unsymmetric-Pattern Multifrontal Method. *ACM Trans. Math. Softw*, 2004.

[9] Timothy Davis. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2006.

[10] Philip Garcia and Katherine Compton. A Reconfigurable Hardware Interface for a Modern Computing System. *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 73–84, April 2007.

[11] Intel. Intel Atom Processor E6x5C Series-Based Platform for Embedded Computing. Technical report, 2011.

[12] C. Nwankpa J. Johnson, P. Vachranukunkiet, S. Tiwari, P. Nagvajara. Performance Analysis of Loadflow Computation Using FPGA. In *Proceedings of 15th Power Systems Computation Conference*, Liege, Belgium, 2005.

[13] Douglas Jones. *Data Pump Architecture Simulator and Performance Model.* Masters thesis, 2010.

[14] ARM Limited. AMBA 4 AXI4-Stream Protocol, 2010.

[15] ARM Limited. AMBA AXI Protocol Specification, 2010.

[16] Inc. MATLAB The Mathworks. Simulink User's Guide.

[17] SPIRAL. DPA Instruction Set Architecture V0.2 Basic Configuration (SPARC V8 Extension), 2008.

[18] Petya Vachranukunkiet. *Power flow computation using field programmable gate arrays.* PhD thesis, 2007.

[19] Matthew A Watkins and David H Albonesi. ReMAP : A Reconfigurable Heterogeneous Multicore Architecture. *Micro*, 2010.

[20] Xilinx Inc. EDK Concepts, Tools, and Techniques: A Hands-On Hands-On Guide to Effective Embedded System Design.

[21] Xilinx Inc. Programmable Logic Design Quick Start Handbook, 2006.

[22] Xilinx Inc. Virtex-II Pro and Virtex-II Pro X FPGA User Guide, 2007.

[23] Xilinx Inc. System Generator for DSP: User Guide. UG640. v12.3., 2010.

[24] Xilinx Inc. Xilinx Virtex-6 Family Overview, 2010.

[25] Xilinx Inc. AXI Reference Guide, UG761, 2011.

[26] Xilinx Inc. LogiCORE IP AXI DMA (v3.00a), 2011.

[27] Xilinx Inc. MicroBlaze Processor Reference Guide, 2011.

[28] Like Yan, Binbin Wu, Yuan Wen, Shaobin Zhang, and Tianzhou Chen. A Reconfigurable Processor Architecture Combining Multi-core and Reconfigurable Processing Unit. *2010 10th IEEE International Conference on Computer and Information Technology*, June 2010.